Supporting Software Evolution in Agent Systems

A thesis submitted for the degree of
Doctor of Philosophy

Khanh Hoa Dam B.Cs., M.App.Sc (IT)
School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

28th August, 2008

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Khanh Hoa Dam
School of Computer Science and Information Technology
RMIT University
28th August, 2008

Acknowledgments

First of all, my deepest gratitude must go to my supervisors, Associate Professor Michael Winikoff and Professor Lin Padgham, for providing me with the wonderful opportunity to complete my PhD thesis. I would particularly like to thank Michael, who patiently guided me through all the steps of my candidature with his enormous amount of constant support. He has always placed substantial faith in my ability, and motivated and encouraged me through the most difficult times. I feel extremely lucky and privileged to work with him and have his excellent guidance.

I am also very grateful to Lin for her invaluable comments and suggestions along the way of doing this research and writing the thesis. I also greatly thank Lin for helping me obtain the scholarship that has been of great assistance to me over the last few years.

Michael and Lin have taught me many valuable lessons about being a good researcher, and the knowledge I gained from them over the last few years is priceless. For all of the things they have done for me that can hardly be expressed in just a few lines here, I sincerely thank once again them both.

I wish to thank RMIT University for providing me with a generous scholarship, that allowed me to pursue my dream of reading for a PhD. I thank the various administrative and technical support staff at RMIT, who have helped me at different stages of my candidature.

I am very fortunate to work in the very active and friendly agent research group at RMIT. I had a great opportunity to advance my presentation skills and obtained many useful feedbacks to improve my work when presenting at the group's weekly meetings. I would like to thank all members of the group for their friendship and support, especially Sebastian Sardina for many stimulating discussions regarding the work in chapter 7, Duc Pham for reading and commenting on a draft of the thesis, and John Thangarajah for assistance in integrating the Change Propagation Assistant with the Prometheus Design Tool.

I owe special gratefulness to my family for continuous and unconditional support, especially to my father for the enormous amount of sacrifices that he has made to allow me to get this far in my education. His courage and gritty determination to overcome various difficult challenges in life have also been inspiring me and pushing me forward. I also thank my beloved wife whose enduring patience, understanding and sacrifices gave me the freedom and energy to concentrate on the research. Finally, this thesis is dedicated to my late mother for her incredible love and sacrifice.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- Khanh Hoa Dam and Michael Winikoff. Cost-based BDI plan selection for change propagation. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 217–224, Estoril, Portugal, May 2008 (work included in Chapters 7 and 9).

- Khanh Hoa Dam and Michael Winikoff. Evaluating an agent-oriented approach for change propagation. In M. Luck and J. J. Gomez-Sanz, editors, *Proceedings of the Ninth International Workshop on Agent Oriented Software Engineering*, pages 61–72, Estoril, Portugal, May 2008 (work included in Chapter 9).

- Khanh Hoa Dam. An agent-oriented approach to support change propagation in software evolution. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Extended Thesis Abstract for Doctoral Mentoring Program, pages 1736–1737, Estoril, Portugal, May 2008 (work included in Chapter 3).

- Khanh Hoa Dam and Michael Winikoff. Generation of repair plans for change propagation. In M. Luck and L. Padgham, editors, *Agent-Oriented Software Engineering VIII*, volume LNCS 4951 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin/Heidelberg, April 2008. ISBN 978-3-540-79487-5 (work included in Chapter 6).

- Khanh Hoa Dam, Michael Winikoff and Lin Padgham. An agent-oriented approach to change propagation in software evolution. In J. Han and M. Staples, editors, *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 309–318. IEEE Computer Society, 2006. ISBN 0-7695-2551-2 (**Best Research Paper award**) (work included in Chapters 3 and 4).

This work was supported by the Australian Research Council under grant LP0453486, in collaboration with Agent Oriented Software.

The thesis was written in the `WinEdt` editor on Microsoft Windows XP Home Edition, and typeset using the LaTeX 2_ε document preparation system.

Dedication

To the loving memory of my mother, Trần Thị Thái Phỉ (1946 – 1996).

Contents

List of Figures

List of Tables

Abstract

Software maintenance and evolution is arguably a lengthy and expensive phase in the life cycle of a software system. A critical issue at this phase is change propagation: given a set of primary changes that have been made to software, what additional secondary changes are needed to maintain consistency between software artefacts? Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software maintenance and evolution. Furthermore, while most of the existing change propagation approaches focus on source code, support for propagating changes in design models has received less attention. As the importance of models in the software development process has been better recognised, a recent research trend is dealing with changes at the level of design models.

The agent paradigm, with its new way of thinking about software systems as a collection of autonomous, flexible and robust agents, offers a promising solution for modelling and implementing distributed complex systems. Agent systems, like conventional software systems, must evolve to meet the ever-changing user requirements and operating environment. If we are to be successful in the development of agent-oriented systems that remain useful after delivery, the research community must provide solutions and insights that will improve the practice of maintaining and evolving agent systems. While a large number of agent-oriented methodologies and techniques have been proposed to support the process of analysing, designing and implementing agent systems, there has been limited work aiming to improve the maintenance and evolution of such systems.

Our objective is to provide tool support for assisting designers in propagating changes during the process of maintaining and evolving models. We propose a novel, agent-oriented approach that works by repairing violations of desired consistency rules in a design model. Such consistency constraints are specified using the Object Constraint Language (OCL) and

the Unified Modelling Language (UML) metamodel, which form the key inputs to our change propagation framework. The underlying change propagation mechanism of our framework is based on the well-known Belief-Desire-Intention (BDI) agent architecture. Our approach represents change options for repairing inconsistencies using event-triggered plans, as is done in BDI agent platforms. This naturally reflects the cascading nature of change propagation, where each change (primary or secondary) can require further changes to be made.

An important piece of our change propagation framework is a new method for generating interactive repair plans from the OCL consistency constraints that restrict a design model. This provides effective automation in terms of reducing the significant effort that the user would require to write such repair plans manually, but importantly, also ensures that the repair plans are correct. Furthermore, typically a given inconsistency will have a number of repair plans that could be used to restore consistency. We propose a mechanism for semi-automatically selecting between alternative repair plans based on a notion of cost, which takes into account cascades (where fixing the violation of a constraint breaks another constraint), and synergies between constraints (where fixing the violation of a constraint also fixes another violated constraint). Our framework is supported by a proof-of-concept tool, called Change Propagation Assistant (CPA), that is integrated into the Prometheus Design Tool. We used the CPA tool in an evaluation in which different change scenarios involving an existing weather alerting system were examined. The results have shown that given a reasonable amount of primary changes, the approach is able to assist the designer by recommending feasible secondary change options that match the designer's intentions. The evaluation has also shown that the approach can, on average, automatically identify the majority of the actions in a given change plan for meeting a particular change request. Finally, we also illustrate that our approach is applicable not only to agent-oriented models (e.g. Prometheus models) but also to object-oriented models in the Unified Modelling Language (UML).

Chapter 1

Introduction

The ever-changing business environment demands constant and rapid evolution of software. Change is inevitable if software systems are to remain useful. Software maintenance and evolution is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified operational environment [Van Vliet, 2001]. This thesis addresses the problem of software maintenance and evolution in the context of agent-oriented development. We especially focus on improving the support for dealing with changes when maintaining and evolving the design of agent-oriented software.

Change is an important part of the software development activity, which can account for a large percentage of the cost of software development. Although a significant amount of work has been done, software evolution and maintenance is still a challenging problem for both research and practice [Bennett and Rajlich, 2000; Rajlich, 2006]. This is due to the lack of theoretical foundations and empirical studies of software evolution and maintenance, as well as to the inherent complexity of software.

To make matters worse, the level of complexity in software systems is growing, which demands new approaches in software engineering to manage this complexity. The emerging agent-oriented paradigm, with its potential to significantly improve the development of high-quality and complex systems, has attracted an increasing amount of interest from the research and business communities [Luck et al., 2005]. A software agent is a piece of software which is situated in an environment, acts on its own and interacts with other similar entities to achieve some design goals [Wooldridge and Jennings, 1995]. An agent also works proactively

to pursue certain goals while, at the same time, it responds in a timely fashion to changes that occur in its environment. This agent view provides a well suited level of abstraction for modelling, an effective way of decomposing, and an appropriate method for dealing with the dependencies and interactions in complex software systems [Jennings, 2001; Wooldridge and Ciancarini, 2001]. Indeed, there have been numerous agent-based applications in a wide variety of domains such as air traffic control [Ljungberg and Lucas, 1992], space exploration [Muscettola et al., 1998], weather alerting [Mathieson et al., 2004], business process management [Burmeister et al., 2008], holonic manufacturing [Monostori et al., 2006; Shen and Norrie, 1999], e-commerce, and information management [Jennings and Wooldridge, 1998; Munroe et al., 2006].

Agent systems, like conventional software systems, undergo changes during their evolution to meet ever-changing user requirements and environment changes. If we are to be successful in the development of agent-oriented systems that remain useful after delivery, the research community must provide solutions and insights that will improve the practice of maintaining and evolving agent systems. A substantial amount of work in agent-oriented software engineering (AOSE) [Jennings and Wooldridge, 1999] has provided methodologies for analysing, designing and implementing software systems in terms of agents [Bergenti et al., 2004; Henderson-Sellers and Giorgini, 2005]. However, there has been very little work on maintenance and evolution of agent-based systems [Dam and Winikoff, 2003; Tran and Low, 2005].

The main purpose of this research is to fill that gap. More specifically, our work tackles the issue of *change propagation*, which is arguably a central aspect of software maintenance and evolution [Buckley et al., 2005; Rajlich, 2006]. A software system consists of various entities (e.g. agents, plans, events, etc. in an agent-based system) and their dependencies (e.g. an incoming message depending on plans to handle it). Different software engineering paradigms or software systems may consist of different entities and dependencies. A dependency between two entities is *consistent* if what an entity provides satisfies the requirements of the other entity. When a developer makes a change in a software system, he/she usually begins with modifying a particular entity of the system. After this initial, primary, change, the entity may no longer interact properly with other entities of the system, because it may no longer satisfy the requirements of the other entities, or it may now require different services from the entities it depends on. This results in *inconsistent* dependencies (inconsistencies for

short) in the software system. The purpose of a *change propagation* process is to reintroduce consistency into the system by keeping track of the inconsistencies and making additional, secondary changes to repair these inconsistencies [Rajlich, 2006]. The secondary changes, however, may themselves introduce new inconsistencies, which may also trigger additional changes and so on.

One the one hand, change propagation is very important in the process of maintaining and evolving a software system. The software maintainer has to ensure that the change is correctly propagated, and that the software does not contain any inconsistencies. Errors and bugs in software are partly due to an unforseen and uncorrected inconsistency. On the other hand, change propagation is a complicated and costly process, especially in complex software systems [Luqi, 1990; Rajlich, 2006; Yau et al., 1978]. As a result, there is an emerging need to have supporting tools and techniques which improve both the efficiency and quality of the change propagation process. Much of the work that has been done in change propagation has been addressing the issue at the code level (e.g. [Gwizdala et al., 2003; Hassan and Holt, 2004; Rajlich, 2001; 1997]). The recent emergence of model driven development (e.g. MDA [Kleppe et al., 2003]) has better recognised the importance of models in the software development process [Mellor et al., 2003; Schmidt, 2006]. Current modelling environments, however, do not adequately address software evolution problems, which leads to an emerging need to deal with software changes at the model level [van Deursen et al., 2007]. As a result, the main focus of our work is to deal with *propagating changes through design models.*

1.1 Research questions

Our approach to change propagation is based on the conjecture that given a suitable set of consistency relationships[1], change propagation can be done by finding and fixing inconsistent relationships in a design model, which are caused by primary changes made to the model. As a result, our research revolves around two main problems: the first is establishing consistency relationships between design entities and models, and the second is representing and implementing a mechanism for fixing inconsistencies caused by changes made to design models. These two problems correspond to our first two research questions:

1. What type of consistency relationships can be derived from design artefacts to support change propagation and how can they be identified and represented?

[1]Consistency relationships refer to some relationship that should hold between model elements.

2. How to effectively represent and implement a mechanism for propagating changes by fixing those consistency relationships when they are violated?

Typically for a given inconsistency there will be multiple ways of fixing it, which leads us to the following questions:

3. (a) What is an appropriate representation for capturing different options of repairing an inconsistency? (b) What kinds of automation can be provided to generate such repair options?

4. How to select between different applicable repair options to fix a given consistency violation? The selection must take into account the potential side-effects, where fixing one inconsistency may inadvertently affect how to fix another one.

Our last research question is pragmatic, and relates to the need for tools to support change propagation in software maintenance and evolution.

5. What type of tool support can be given to designers to assist in the process of understanding and modifying an existing agent system? In our view, a tool cannot fully automate change propagation because a tool cannot make decisions involving trade-offs and design styles where human intervention is required. However, the tool can help the designer by presenting feasible change propagation options.

1.2 Research outcomes and main contributions

This thesis makes several contributions to the state of the art. Firstly, we have developed an agent-based framework to propagate changes by fixing inconsistencies in design models. Consistency relationships are defined as constraints that describe conditions that all design models must satisfy for them to be considered valid, e.g. syntactic well-formedness, coherence between different diagrams, and even best practices. Such consistency constraints are specified on the Unified Modelling Language (UML) [Object Management Group, 2005] metamodels using the formal language Object Constraint Language (OCL) [Object Management Group, 2006] (research question 1).

Secondly, we have adopted the well-known Belief-Desire-Intention (BDI) architecture to represent and implement the underlying change propagation mechanism (research question

2). Repair options are represented as BDI plans whilst fixing a constraint is considered as a goal/event (research question 3a). The use of BDI-style, event-triggered plans, matches well with the cascading nature of change propagation, where a change can cause other changes to be made. Further, there are usually many ways of fixing a given inconsistency, and this is naturally captured using multiple plans that respond to a given event. Such *abstract repair plans* are a way to reasonably enumerate the otherwise large number of concrete ways of fixing inconsistencies. Although we do not use the full capabilities of a BDI agent, these two properties of change propagation make the use of BDI plans natural and, we believe, well-motivated.

Thirdly, we have also developed a repair plan generator that is able to automatically produce repair plans for a given constraint (research question 3b). This significantly reduces the substantial amount of time and effort that the user would have to spend on writing such repair plans manually. Another key consequence of generating plans, rather than writing them manually, is that by careful definition of the plan generation scheme, it is possible to guarantee the completeness and correctness of the generated plans.

Fourthly, our research has directly addressed the general problem of (applicable) BDI plan selection in the context of change propagation (research question 4). We dealt with this issue by defining a suitable notion of repair plan cost that incorporates both conflict between plans, and synergies between plans. We then developed an algorithm, based on the notion of cost, that finds cheapest options and proposes them to the user.

Finally, our last contribution is the implementation of the Change Propagation Assistant (CPA) tool that demonstrates the applicability and practicality of our approach (research question 5). The tool is integrated with the Prometheus Design Tool (PDT)[2] [Padgham et al., 2005], a modelling tool that supports the Prometheus methodology for building agent-based systems. We then performed an empirical evaluation to assess the efficiency and effectiveness of our change propagation framework generally and the CPA tool in particular.

Although the CPA tool, a prototypical implementation of our change propagation framework, is specific to the Prometheus methodology, we believe that the framework is generic in which it can be applied to a range of design types. In fact, we have shown that our approach is applicable not only to agent-oriented models (e.g. Prometheus models) but also to object-oriented design models such as UML models. Our work aims not only to provide

[2]http://www.cs.rmit.edu.au/agents/pdt

solutions for software maintenance in agent-oriented software engineering but also to apply agent technology to the problem of software maintenance in a broader context.

1.3 Existing work

There has been very little work on the field of maintenance and evolution of agent systems. Although there have been a number of agent-oriented methodologies proposed in the past few years, previous studies [Dam and Winikoff, 2003; Tran and Low, 2005] have shown that none of the prominent methodologies explicitly and extensively cover the maintenance phase. In particular, we are not aware of any existing work on dealing with the critical issues of change propagation in maintaining and evolving agent-based design models.

On the other hand, software maintenance has been an active area of research in the field of software engineering. There has been a lot of interest in addressing the issue of assisting software engineers to deal with software changes. In particular, change impact analysis has been extensively investigated but most of the work has focused on source code. Many of the impact analysis approaches are discussed in [Arnold and Bohner, 1996] and are typically used to assess the extent of the change, i.e. the artefacts, components, or modules that will be impacted by the change, and consequently how costly the change will be. Although these approaches are very powerful, they do not readily apply to design models [Egyed, 2007].

Model consistency checking is an area clearly related to change propagation and there has been a wide range of work in this area (e.g. [Bodeveix et al., 2002; Breu et al., 1997; Engels et al., 2002; Ivkovic and Kontogiannis, 2004; Kuzniarz et al., 2002; 2003; Van Der Straeten et al., 2003]). A change propagation framework can be built on top of existing consistency checking approaches. However, it should be noted that the iterative nature of cascading changes ideally requires incremental consistency checking as proposed in ArgoUML[3] and xlinkit [Nentwich et al., 2002]. Advanced event-driven consistency checking approaches such as Egyed's [2006] can be integrated in order to improve the consistency checking capability of a change propagation framework.

There have been several works that specifically target fixing inconsistencies in design models. The work by Nentwich et al. provides a framework which automatically derives a set of repair actions from the constraint by analysing consistency rules expressed in first-order logic and models expressed in xlinkit [Nentwich et al., 2002; 2003]. Their work considers

[3]http://argouml.tigris.org/

only a single inconsistency and consequently does not explicitly address dependencies among inconsistencies and potential consequences of repairing them, e.g. fixing one constraint can repair or violate others. However, their work proposes a fundamental idea that can be further extended: that ways of fixing inconsistencies can be represented as abstract repair actions.

Recently, Egyed proposed an approach based on fixing inconsistencies in UML models [Egyed, 2007]. The approach uses model profiling to locate possible starting points for fixing an inconsistency in a UML model. He also tried to use model profiling to predict the side-effects of fixing an inconsistency. His work, however, treats a constraint as a black box and consequently does not benefit from the knowledge of the constraints. Similarly the work of Briand et al. also looks at how to identify impacted entities during change propagation using UML models [Briand et al., 2006]. It defines specific change propagation rules (expressed in OCL) for a taxonomy of changes. However, their approaches do not provide options to repair inconsistencies as a way of propagating changes, but only suggest starting points (entities in the model) for fixing the inconsistency.

1.4 Thesis structure

The remainder of this thesis is structured as follow.

- Chapter 2: "Background" provides background material and existing work related to this thesis. This includes an overview of software maintenance and evolution with a special focus on change propagation. This chapter also provides background information which details the concepts of agents (especially BDI agents), agent oriented software engineering (AOSE), and the Object Constraint Language (OCL).

- Chapter 3: "Change Propagation Framework" provides a description of our approach to the issue of change propagation in maintaining and evolving agent systems.

- Chapter 4: "Case Study 1: Prometheus" describes how our approach can be applied to the Prometheus methodology, a prominent agent-oriented methodology.

- Chapter 5: "Case study 2: UML" provides a description of a small case study for the purpose of illustrating how our approach can support change propagation in UML object-oriented design models.

- Chapter 6: "Plan Generation" presents in detail one of the key and novel components of our change propagation framework: a repair plan generator, which takes a given OCL constraint and automatically produces repair plans that can fix the constraint.
- Chapter 7: "Plan Selection" addresses an important question that needs to be answered as part of our change propagation framework, which is how to select between different applicable (repair) plan instances to fix a given constraint.
- Chapter 8: "Implementation" provides a description of a prototype tool support for change propagation, called Change Propagation Assistant, that we have developed.
- Chapter 9: "Evaluation" presents, discusses and analyses the results of an empirical evaluation of our approach.
- Chapter 10: "Conclusion" revisits our research questions and highlights our major contributions as well as raises suggestions for future research.

Chapter 2

Background

The main purpose of this chapter is to provide some insight into the problem that this research attempts to tackle. The first part of this chapter provides some background on software maintenance and evolution, with a special focus on change propagation, the particular issue of software maintenance that we aim to address. More specifically, in section 2.1 we give an overview of the history of software maintenance and evolution, and some key areas for research. In addition, we present a classification of changes, the underpinning force of software maintenance and evolution. Furthermore, we explain the role of change propagation in the context of the software maintenance process. We also describe a range of major approaches and techniques for change propagation.

The second part of this chapter (section 2.2) gives some background on the agent-oriented approach, including an introduction to agents and a brief description of the famous Belief-Desire-Intention (BDI) agent architecture. In this section, we also give an overview of Agent-Oriented Software Engineering (AOSE) and briefly explain its potential for being an efficient and powerful software engineering approach. We then highlight some of the key ideas, research questions, and current challenges in AOSE.

The final section of this chapter (section 2.3), briefly describes the Object Constraint Language (OCL), which is a standard language for describing rules that apply to UML models. We briefly explain how OCL is used, including its syntax, using a simple example. Finally, we present a description of current tool support for OCL.

2.1 Software maintenance and evolution

The development of software involves a number of activities such as requirement specification, architecting, design, implementation, testing, and deployment. As software systems are continuously evolving and changing, maintaining and enhancing them is an inevitable and challenging activity. More and more software is being built every day and more organisations are now strongly dependent on their software systems. Such software systems are critical business assets and a substantial investment must be put into system change to maintain the value of these assets. As a result, software maintenance and evolution has become an active area of research and practice.

2.1.1 Overview

> *"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."* – IEEE Standard 1219 for Software Maintenance [IEEE, 1999]

During more than 60 years of modern computing, there have been major changes in the interpretation and importance of software maintenance [Bennett, 2005]. In the early days, the focus was mainly on writing new software and consequently software maintenance did not attract much attention. As time went on, more software products were created and therefore a steadily increasing amount of effort went into the support of these products. As a result, in the late 1960s and early 1970s software maintenance began to be recognised as an important activity after software delivery. The decade of the 1980s witnessed the difficulties of modifying legacy systems developed based on old architectures (which had been used for the previous two decades) in order to meet new business requirements [Lientz and Swanson, 1980]. Since then, there has been a strong need for software systems to be evolved to match with the ever-changing business environment [Parnas, 1994].

The notion of software evolution also originated in the late seventies, when Manny Lehman started to develop his, now well-known, *laws of software evolution* based on his empirical studies performed on IBM's OS 360 operating system [Lehman, 1980; Lehman and Parr, 1976]. Lehman's laws arguably laid an important foundation for the study of software maintenance and evolution. In fact, it was possibly the first time that the term *software evolution* was intentionally used to emphasize the difference with the post-deployment activity

of software maintenance [Mens, 2008]. There are eight of Lehman's laws including continuing change, increasing complexity, self regulation, invariant work rate, conservation of familiarity, continuing growth, declining quality, and feedback system. In particular, Lehman's laws of evolution stress that successful software systems are committed to evolve over time since they "*operate in or address a problem or activity of a real world*". Lehman's laws also suggest that product and process attributes such as size, time between releases, and the number of reported bugs are almost invariant for each release (i.e. self regulating), or that the incremental change in each release in statistically invariant over the lifetime of a system (i.e. conservation of familiarity). His later studies [Lehman, 1996; Lehman and Belady, 1985; Lehman et al., 1997] concerning other software systems confirm his original findings.

Changes relating to bug fixing or error correction were the typical image of software maintenance in the early days. By contrast, evolutionary change has been the essence of software development since the 1990s. In fact, enhancement and evolution are increasingly needed (and even inevitable) to accommodate business changes [Bennett and Rajlich, 2000]. The definition of software maintenance has recently extended: "*Software maintenance is the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage*" (according to the ISO/IEC and IEEE Standards 14764-2006 [IEEE, 2006]). From this software engineering perspective, the terms software evolution and software maintenance are virtually synonymous[1], referring to the same activity in the software life cycle and share the same economic, organisational and technical concerns [Mens, 2008]. The only difference between the two terms is that evolution has more appealing aesthetics whilst maintenance is usually associated with negative connotation. Maintenance seems to lead to an incorrect assumption that the software system becomes lower in quality. In reality however, various studies and surveys [Abran and Nguyen, 1993; Bennett and Rajlich, 2000; Pigoski, 1996; Takang and Grubb, 2003] have shown that more than 80% of the total maintenance effort is spent on adapting the software to meet changes in the environment or user needs.

A substantial percentage – as much as two-thirds – of the cost of any software are attributed to its maintenance, making software maintenance a critical technical and economic factor [Koskinen, 2004; Lientz and Swanson, 1980; Seacord et al., 2003; Van Vliet, 2001]. As a result, there has been a significant amount of research in this area. Approaches have been

[1]The two terms are also used interchangeably in this dissertation.

proposed to support crucial activities in software maintenance such as program comprehension and reverse engineering [Müller et al., 2000], restructuring [Fowler and Beck, 1999; Mens and Tourwé, 2004], re-engineering [Miller, 1998], impact analysis [Arnold and Bohner, 1996], and management processes [Polo et al., 2003; Takang and Grubb, 2003]. Those approaches will be discussed later in the chapter. However, dealing with software evolution remains one of the most challenging issues in mainstream software engineering [Bennett and Rajlich, 2000; Mens, 2008]. In order to gain more understanding about the problem, the next section serves to identify various types of changes that motivate maintenance and evolution activities.

2.1.2 Classification of changes

Software change is the essence of software maintenance and evolution. An important step towards understanding the importance of maintenance and its implication on the costs and the quality of software systems is identifying the reasons that drive the need for change. As a result, there has been a range of work in defining a taxonomy for software maintenance.

Nearly three decades ago, Swanson [1976] proposed a software maintenance typology in terms of distinguishing the purposes of three activities: *perfective*, *adaptive* and *corrective* maintenance. Corrective maintenance introduces changes to remove bugs in the software. Adaptive maintenance involves changing the system so that it can continue to work in a changed environment, for instance modifying the system to make it run on a new operating system or hardware platform. Finally, perfective maintenance aims to add, delete or modify the system's functionalities resulting from user needs. Recent studies [Abran and Nguyen, 1993; Bennett and Rajlich, 2000; Pigoski, 1996; Takang and Grubb, 2003] have shown that most maintenance tasks involve perfective and adaptive maintenance, rather than error correction.

Based on the above taxonomy, software maintenance is mostly used to improve the quality of a software system. On the other hand, software changes usually lead to the increased complexity of software and make it harder to maintain. In fact, Lehman's second law of software evolution states that "*as an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure*" [Lehman, 1980]. Accordingly, a new type of maintenance, namely *preventive* maintenance, has been introduced to indicate changes that improve the maintainability of software, providing a basis for future enhancements [Lientz and Swanson, 1980]. However,

the term preventive maintenance is confusing since it is also viewed as modifications of a software product after delivery to detect and correct latent faults in the software product before they become effective faults [Chapin, 2000; Kajko-Mattsson, 2000]. The ISO standards [ISO/IEC 14764, 1999] adopt this latter view of preventive maintenance together with the three classical types of software maintenance proposed by Swanson [1976].

There has also been other work that extends the above types of software maintenance. For instance, Chapin et al. [2001] refined Swanson's typology into an evidence-based classification of 12 different types of maintenance activities: *evaluative*, *consultive*, *training*, *updative*, *reformative*, *adaptive*, *performance*, *preventive*, *groomative*, *enhancive*, *corrective* and *reductive*. These types are determined based on the maintenance and evolution activities performed on different parts of the system. Specifically, updative and reformative maintenance involve changes made to the documentation. In addition, there are maintenance activities that affect the source code in terms of either changing the functionalities (enhancive – functionalities added or replaced; corrective – functionalities fixed; and reductive – functionalities removed), or not changing the functionalities (adaptive – use new resources or technologies; performative – alter system performance; preventive – avoid future maintenance; and groomative – make the system more maintainable). Finally, Chapin et al. [2001] also identified maintenance activities that relate to the supporting aspect of the software rather than the software itself (i.e. no changes made to the software). These include using the software for stakeholder training, as a basis for consultation or for evaluating the software.

These works classify software maintenance and evolution activities on the basis of their purpose, i.e. why software changes take place. On the other hand, Buckley et al. [2005] recently proposed a taxonomy of software change that focuses on characteristics of software change mechanisms and the factors that influence these mechanisms. This addresses several questions including: when is a change made? (e.g. time of change, change history, and change frequency), where is a change made? (e.g. software artefacts, granularity of the change in terms of the scale of the artefacts to be changed, change impact, and change propagation), what is being changed? (system properties such as availability, openness, and safety), and how is the change accomplished? (e.g. degree of automated support for change, and degree of formality of a change support mechanism).

2.1.3 Change propagation in a change mini-cycle process

Regardless of what types of maintenance activities are performed, the outcome of software maintenance and evolution is changes made to the existing system. There are many aspects related to software change ranging from technical issues to managerial concerns. In order to place the issue of change propagation, which is the main focus of our work, in the proper context, we describe here the so-called *change mini-cycle* (see figure 2.1). This change process was first developed by Yau et al. [1978] in the late seventies and is still widely accepted [Bennett and Rajlich, 2000; Mens, 2008; Rajlich, 1999]. It is noted that this process addresses the technical concerns rather than the managerial aspects of software change. In the context of software development, each change mini-cycle can be considered as a incremental change [Rajlich, 2001], which is popularly used in agile software development approaches (e.g. the well-known eXtreme Programming [Beck, 1999]).

This process is triggered by a request which contains the specification of the change. The request for change can have several forms, for example a bug report requesting to remove a fault in the software or an enhancement request for adding a new functionality. The change request can be generated by various actors such as a user, a developer, or a tester. If the request is accepted, the process moves into the planning phase. The main purpose of the planning phase is to perform some analysis and planning before deciding whether and how the change should be implemented. This phase consists of two major activities: *program comprehension* and *change impact analysis*.

Program comprehension

Program comprehension or program understanding refers to the processes used by software developers to understand programs or software usually before modifications are made to software. Program comprehension is a costly process, to which can be attributed more than half of the cost of maintenance [Fjeldstad and Hamlen, 1982] (as cited in [Bennett and Rajlich, 2000]). Program comprehension can involve activities such as reading (source code, documentation, UML diagrams, comments), running the software and analysing its execution trace, or interviewing users and developers. A useful technique that is used to comprehend software is reverse engineering, which refers to "*the process of analyzing a subject system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction*" [Chikofsky and Cross, 1990].

Reverse engineering activities include deriving information from available software artefacts (e.g. source code) and converting it into higher-level, more abstract, human understandable representations (e.g. design models).

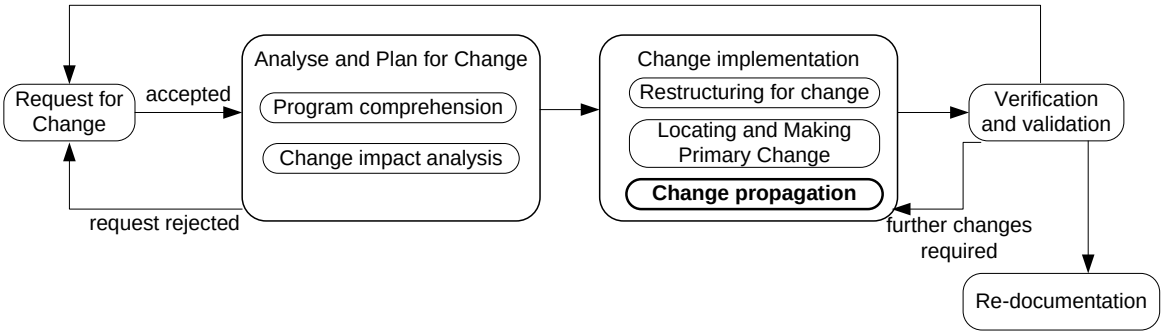


Figure 2.1: A change mini-cycle process (adapted from [Mens, 2008; Rajlich, 1999; Yau et al., 1978])

Change impact analysis

Another important activity in the planning phase is change impact analysis, which is defined as "*identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change*" [Bohner and Arnold, 1996]. The process of change impact analysis contains two major steps. First, the analyst examines the change request and identifies the software items (e.g. the artefacts, components, or modules) initially affected by the change. Next, the analyst identifies other items in the software that apparently have dependency relationships with the initial ones, and forms a set of impacts. Those impacted items also relate to other items and thus the impact analysis continues this process until a complete graph is obtained beginning at the selected items and ending with items on which nothing else depends.

Existing techniques for change impact analysis tend to fall into two groups: static and dynamic analysis. Static impact analysis techniques (e.g. [Ryder and Tip, 2001] or a number of techniques reported in [Arnold and Bohner, 1996]) usually perform either program slicing [Weiser, 1981] or graph traversals to compute impact sets in terms of collecting data related to potential impacts. These techniques are said to be conservative in that they consider all possible program inputs and behaviours. Results produced by static analysis may have enormous impact sets, which are sometimes unnecessary or even too large to be of practical

use [Breech et al., 2006]. Bohner [2002] proposed a range of guidelines to help find solutions to these issues including the use of impact semantics and structural constraints (e.g. distance between a change and an impact) to structure static analysis results. In contrast, dynamic analysis techniques (e.g. [Apiwattanapong et al., 2005; Breech et al., 2005; Law and Rothermel, 2003]) compute impacts using data obtained from executing a program. Dynamic analysis results are more practically useful since they better reflect how the system is actually being used, and consequently do not have computed impacts derived from impossible system behaviour. However, due to their dependency on the inputs used to execute the program, the results produced by dynamic analysis will not include impacts for parts (e.g. functions) of the program that are not executed. As a result, recent approaches (e.g. [Breech et al., 2006]) aim to improve the precision of dynamic impact analysis results.

Although a large portion of change impact analysis techniques focus on source code analysis, there are a few approaches that target models. The work in [Kung et al., 1994] addresses how change impact analysis can be performed from a class diagram. They also discussed how object-oriented properties such as inheritance, encapsulation, polymorphism and so on affect their impact analysis. They proposed an algorithm to identify the impacted parts of the system by calculating the delta of two versions of software. Their analysis, however, only used static information, i.e. the class diagram. In addition, there has been some recent work aiming at (semi-)automated support for impact analysis of UML models. For example, von Knethen and Grund [2003] developed a tool environment called QuaTrace that semi-automatically identifies impacts on UML models when system requirements (e.g., use case descriptions) undergo changes. Their approach is based on the establishment of traceability links between textual descriptions of use cases and UML model elements. The work of Briand et al. [2006] also computes change impacts for actions for UML models but takes a different approach. They identified specific impact analysis rules (defined with the Object Constraint Language [Object Management Group, 2006]), which are used to determine model elements that are directly or indirectly impacted by the changes. They also proposed a measure of distance between changed model elements and impacted elements in order to sort the resulting impact sets according to their probability of occurrence.

In summary, change impact analysis aims to assess the extent of the change, i.e. the artefacts, components, or modules that will be impacted by the change, and consequently predicts the cost and complexity of the change. Change impact analysis plays a major part

in establishing the feasibility of a change in terms of determining whether the change is to be undertaken. If the change is rejected, then the request is returned to its origin for revision or more discussion. Otherwise, the change process enters the next phase which involves the implementation of the change.

Change propagation

Change implementation leads to actual changes made to the software system. This process usually starts with locating and making some primary changes to certain parts of the system. The primary changes may make those parts no longer interact properly with other parts of system and consequently may cause inconsistencies in the system. For example, adding a message to a sequence diagram might break a consistency rule that messages must correspond to operations in the class of the message's receiver. Therefore, additional, secondary changes are made to reintroduce consistency into the system. As in our earlier example, the inconsistency can be fixed by adding an operation to the appropriate class. The secondary changes, however, may introduce new inconsistencies, which may also trigger additional changes and so on[2]. Determining and making these secondary changes is termed *change propagation*, which is regarded as the central activity within the change implementation phase [Buckley et al., 2005]. Activities in the previous planning stage, change impact analysis in particular, also adds valuable information to change implementation by suggesting which parts of the software need to be changed or are impacted by changes. In section 2.1.4 we discuss different change propagation techniques in more detail.

Restructuring

During the change implementation phase, there may be situations where the change cannot be easily and directly implemented. For instance, a change may involve many components in the system if the concepts relevant to the change are widely spread over those components. A well-known approach to deal with these issues is restructuring. *Software restructuring* is the process of improving the logical structure of existing software systems in terms of making it easier to understand and to change, or less susceptible to error when future changes are introduced [Arnold, 1986]. For example, the solution to the issue of a change's relevant

[2]This phenomenon is usually referred to as the ripple effect or cascading nature of change propagation [Yau et al., 1978].

concepts being delocalized is to restructure the program so that those concepts are localized in one location but the behaviour of the program is still preserved. Other examples of software restructuring include renaming variables, moving expressions, improving coding style, relocating functional components into different modules and so on. Cohesion (i.e. the relatedness of components within a module) and coupling (i.e. connectedness of modules) are two common restructuring criteria [Kang and Bieman, 1999].

In an object-oriented context, software restructuring is commonly referred to as *refactoring* [Fowler and Beck, 1999; Opdyke, 1992]. Since Fowler's well-known book on refactoring object-oriented programs with his introduction of refactoring patterns and processes, there has been a substantial amount of interest in refactoring. A recent survey by Mens and Tourwé [2004] gives an extensive overview of existing refactoring approaches from a range of perspectives such as the degree of support for refactoring activities, techniques and formalisms that are used for supporting these activities, and the types of software artefacts that are being refactored. In addition, there has been an emerging number of refactoring support tools (e.g. Eclipse[3], IntelliJ IDEA[4], RefactorIT[5], Borland JBuilder[6], Condenser[7], and jCOSMO[8]). However, an recent evaluation study in [Mealy and Strooper, 2006] has shown that existing tools for refactoring (those that we listed earlier) do not provide sufficient support for the entire refactoring process as defined by Fowler and Beck [1999], and Mens and Tourwé [2004]. In addition, the evaluation also identifies some issues relating to the maturity and usability of existing tool support. Furthermore, most existing refactoring tools focus on source code, but support for refactoring design models has received less attention. As models have received better recognition, a recent research trend is model-driven software refactoring. Model refactoring is a model transformation that preservers some behavioural properties of a model [Porres, 2003]. A number of approaches and tools have been proposed to deal with refactoring UML models, e.g. [Astels, 2002; Boger et al., 2003; Straeten and D'Hondt, 2006; Zhang et al., 2005]. However, there is still a need to have formalisms, processes, methods and tools that deal with refactoring in a more consistent, generic, scalable and flexible way [Mens, 2008; Mens and Tourwé, 2004].

[3] http://www.eclipse.org
[4] http://www.jetbrains.com/idea
[5] http://www.refactorit.com
[6] http://www.borland.com/us/products/jbuilder
[7] http://condenser.sourceforge.net
[8] http://www.cwi.nl/projects/renovate/javaQA/intro.html

Verification and validation

After the change is implemented, validation and verification are performed to make sure that (a) the system integrity has not been compromised (e.g. no new errors are introduced), and (b) the change is correctly implemented, i.e. the requirement described in the change request is satisfied (e.g. the bug is fixed or new functionality is added). Software testing techniques [Ammann and Offutt, 2008] are used in this phase. For instance, regression testing which involves repeating all existing test cases for the modified system is usually used to verify that the change did not have any undesirable effects. A detailed survey of research in regression techniques can be found in [Rothermel and Harrold, 1998]. During the validation and verification phase, if problems are discovered, the change may need to be re-implemented.

Re-documentation

The final activity of the change process is updating the program documentation to match with changes that have been implemented in the program. For instance, if changes are made to source code, the design needs to be updated to reflect those changes. Depending on the type of changes, other documentation such as user guides may also need to be updated. In addition, Bennett and Rajlich [2000] point out that missing or incomplete program documentation can be updated at this point using the outcome of the program comprehension performed earlier.

2.1.4 Related work on change propagation

We have previously examined the change mini-cycle process which consists of a number of important activities related to the change process including program comprehension, change impact analysis, restructuring, and change propagation. We also highlighted the key approaches that have been proposed to improve the support for these change activities. Within this context, the focus of our work is, however, on change propagation. In this section we discuss existing related work in dealing with the issue of change propagation in software maintenance.

Formalisation of change propagation process

An important activity of change propagation is identifying what are the other parts in the system that need secondary changes. There has been a range of work addressing this issue by formally modelling the change propagation process. Regarding this aspect, change impact analysis and change propagation share several key ideas. A program is usually formalised using graphs where the nodes are the software components, and the edges are dependency relationships between them [Chaumun et al., 2002; Deruelle et al., 2001; Kung et al., 1994; Luqi, 1990; Rajlich, 1997; Yau et al., 1988]. In order to build such a graph, the dependencies in the program must be known. Dependency analysis is one of the impact-analysis techniques to detect and capture information relating to data, control, and component dependencies. Such dependency information is used as a starting point to formally model the process of change propagation. A number of formal models of change propagation reviewed in [Arnold and Bohner, 1996; 1993] use code dependencies and algorithms such as slicing and transitive closure to propose several tools and techniques supporting dependency analysis in change propagation.

One important work involving the formalising of change propagation which also uses graphs as an underlying representation was Rajlich's in the late nineties [Rajlich, 1997; 1999; 2000]. Compared with earlier work, Rajlich's model is more complete in terms of reflecting various strategies of propagation. Rajlich's model uses graph rewriting techniques [Métayer, 1998; Rajlich, 1977] to formalise the change process. In Rajlich's model, the change process is viewed as the evolution of the dependency graph (called an evolving interoperation graph), which is modelled as a sequence of snapshots. Each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent. An evolving interoperation graph contains a set of interoperations which are pairs of interacting software components (e.g. function calls, data flows, use of shared resources, etc.). The changes are propagated through interoperations from one component to the next by identifying and marking inconsistent interoperations. Rajlich then proposes several strategies of visiting inconsistent interoperations such as strict change-and-fix, random change-and-fix, and top-down process. Change-and-fix process visits all the inconsistent components until all of them are resolved. This can lead to components being visited several times and an infinite process may occur when a change is propagated in a cycle. Meanwhile, top-down change propagation visits from the top components (i.e. components which do not

support any other components) down to supporting components. This top-down process (also referred to as Methodology of Software Evolution (MSE) by Rajlich) is more predictable but is based on the assumption that there are no loops in the component dependencies. Rajlich's models have been implemented as a prototype tool called "Ripples 2", which supports both the change-and-fix and the MSE processes of change propagation.

Rajlich's formal model of change propagation has been the foundation of other work. For instance Deruelle et al. [2001] extend the graph model with typed nodes and edges and constraints representing the invariant properties of the software. They then proposed an expert system that contains rules (derived from the constraints) that drive the process of visiting components that need changing. On the other hand, Hassan and Holt [2004] use the intuition and ideas from Rajlich's work to model a simplified change process. They then designed and validated a set of change propagation heuristics based on data collected from open source projects. Those historical heuristics are used to drive the change propagation process by applying changes to all entities which frequently (e.g. 80% of the time) co-changed with the changed entity in the past. Their empirical study has shown that change history based heuristics outperform other types of heuristics that use, for example, dependency information in a number of applications. Recently, Malik and Hassan [2008] argued that those heuristics are static in that they do not adjust over time or adapt to a certain changed entity. As a result, in order to improve the overall performance of change propagation heuristics, they proposed a set of heuristics that can adapt to the current state of a software lifecycle (e.g. new development vs. maintenance), and to the different characteristics of the various entities in the software.

Inconsistency-based change propagation

As the process of change propagation aims to maintain consistency within software, another (more direct) approach to deal with this issue is detecting and resolving inconsistencies caused by changes. Differing from work that we presented earlier, which mainly focus on source code, consistency maintenance approaches address a wide range of software artefacts including design, specifications, documentation, source code, and test cases. Various techniques and methods have been proposed in the literature to address different activities of the consistency management process including: detecting overlaps between software models, detecting inconsistencies, identifying the source, the cause and the impact of inconsistencies,

and resolving inconsistencies [Spanoudakis and Zisman, 2001].

There has been recent work aiming to address a range of software artefacts (e.g. [Reiss, 2005]). These efforts mostly revolve around the development of a software development environment that integrates different phases of a software life-cycle including maintenance.

The influential *Viewpoints* framework [Finkelstein et al., 1992; 1994] supports the use of multiple perspectives in system development by allowing explicit "viewpoints" containing partial specifications, which are expressed and developed using different representation styles and development strategies. In their framework, inconsistencies arising between individual viewpoints are detected by translating into a uniform logical language. Such inconsistencies are resolved by having meta-level inconsistency handling rules. A number of approaches have been proposed to deal with inconsistencies and change management in requirement engineering [Ghose, 2000; Krishna et al., 2009; Tsai et al., 1992; van Lamsweerde et al., 1998]. They focus on resolving inconsistencies between requirements in specifications using a formal approach in which specifications are represented as logics. There is also work that deals with only design models. As UML has become the de facto notation for object-oriented software development, most research work in consistency management has focused on problems relating to consistency between UML diagrams and models [Elaasar and Briand, 2004; Kuzniarz et al., 2002; 2003]. Such approaches have been advocated with the recent emergence of model-driven evolution [van Deursen et al., 2007]. Several approaches strive to define fully formal semantics for UML by extending its current metamodel and applying well-formedness constraints to the model [Bodeveix et al., 2002; Breu et al., 1997]. Other approaches transform UML specifications to some mathematical formalism such as Petri-Nets [Engels et al., 2002], or Description Logic [Mens et al., 2005; Van Der Straeten et al., 2003]. The consistency checking capabilities of such approaches rely on the well-specified consistency checking mechanism of the underlying mathematical formalisms. However, it is not clear to what extent these approaches suffer from the traceability problem: to what extent can a reported inconsistency be traced back to the original model. Furthermore, the identification of transformations that preserve and enforce consistency still remains a critical issue at this stage [Engels et al., 2002].

Recently, Egyed [2006] proposed a very efficient approach to check inconsistencies (in the form of consistency rules) in UML models. His approach scales up to large, industrial UML models by tracking which entities are used to check each consistency rule, and then using this

information to determine which rules might be affected by a change, and only re-evaluate these rules. From a different direction, Blanc et al. [2008] argued that earlier work has a major drawback in terms of dealing with only structural rules (which constrains the consistency of a model), and not addressing methodological rules (which specify consistency with respect to the overall process of constructing a model). Those two types of rules are both equally important and complementary since structural rules are imposed on model states whereas methodological rules restrict model changes [Spanoudakis and Zisman, 2001]. Blanc et al. then proposed a framework that is able to express both structural and methodological rules and detect their violation. Their framework also supports inconsistency detection for rules that are expressed across a number of models which have different metamodels.

There are approaches that go further than detecting inconsistencies. Several approaches provide developers with a software development environment which allows for recording, presenting, monitoring, and interacting with inconsistencies to help the developers resolve those inconsistencies [Grundy et al., 1998]. Other work also aims to automate inconsistency resolution by having pre-defined resolution rules (e.g. [Liu et al., 2002]) or identifying specific change propagation rules for all types of changes (e.g. [Briand et al., 2006; Han, 1997]). Such rules can be formally defined following a logic-based approach (e.g. Liu et al. [2002] used Java Rule Engine JESS, or Mens et al. [2005] used Description Logic, or a graph-based approach such as the graph transformations used in [Mens and Van Der Straeten, 2007; Mens et al., 2006]). However, these approaches suffer from the correctness and completeness issue since the rules are developed manually by the user. As a result, there is no guarantee that these rules are complete (i.e. that there are no inconsistency resolutions other than those defined by the rules) and correct (i.e. any of the resolutions can actually fix a corresponding inconsistency). In order to deal with this issue, Nentwich et al. [2003] has proposed an approach for automatically generating repair options by analysing consistency rules expressed in first order logic and models expressed in xLinkIt [Nentwich et al., 2002]. However, they did not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. In other words, their work considers repair actions as independent events, and thus does not explicitly deal with the cascading nature of change propagation.

Those previous works do not address dependencies between inconsistencies and the potential side effects of fixing them (i.e. repairs that inadvertently and adversely have influence

on how to fix other, related inconsistencies), and consequently do not reflect the cascading nature of change propagation. Recently, Egyed [2007] proposed an approach based on fixing inconsistencies in UML models which uses model profiling to locate choices of starting points for fixing an inconsistency in a UML model. His work does not analyse consistency rules but rather observes their behaviour during evaluation (i.e. model profiling) and consequently considers consistency rules as black-boxes. More specifically, he used model profiling to determine which model elements and fields are accessed during the evaluation of a consistency rule. These pairs of a model element and field form so-called scope elements for that given rule instance. This model profiling data (i.e. scope elements) is useful because it indicates the locations (i.e. the model elements) that potentially fix a given inconsistent rule. To be complete, the scope of a rule also includes what he called "back-pointing" scope elements, which have fields relating to one another due to the bidirectional relationships in UML (e.g. a receiver of a message is an object that has the message as an incoming message). He argued that if there are common scope elements among inconsistencies (i.e inconsistent rules), there exists a dependency among these inconsistencies, which implies an opportunity for fixing them by making a single design change. On the other hand, if a dependency among inconsistencies does not exist, then these inconsistencies cannot share any location for fixing them. He also tried to predict the side-effects of fixing an inconsistency by marking scope elements with different annotations, which indicates for example which scope elements are guaranteed to not affect another rule, or which affect only inconsistent rules. Egyed's approach was empirically evaluated on 48 UML models which had different sizes and were from various domains. He used 34 types of rules addressing UML class, sequence, and statechart diagrams. For the purpose of profiling, these rules were instantiated more than 400,000 times for all models. The results showed that his approach is able to identify the dependencies among inconsistencies and locate the common choices (i.e. incorrect model elements) for repairing them. Another benefit of the use of model profiling is the performance gain. Indeed, as empirically validated, his approach scales relatively well to large models. The main limitation in his work, however, is that his approach does not provide options (i.e. how) to repair inconsistencies, but only suggests starting locations (entities in the model) for fixing the inconsistency. Furthermore, the list of locations identified by his approach does contain false positives, i.e. locations for which there is no actual concrete fix that resolves a given inconsistency. Finally, as he also acknowledged, the performance of his approach was evalu-

ated based on the assumption that non-scalable rules (e.g. rules that access a large number of model elements) are excluded, which may not always be the case in practice.

Change propagation and model transformation

Change propagation takes place not only within a model but also between models at different levels of abstractions or expressed in different languages (e.g. design and source code, or UML models and ER models). In the context of model driven software development [Mellor et al., 2003], especially the Model Driven Architecture (MDA) [Kleppe et al., 2003], target models are obtained from source models using model transformations [Sendall and Kozaczynski, 2003]. In this context, one of the issues related to model evolution is that changes in the source model (e.g. design) should be propagated to the target model (e.g. source code) and vice versa. There are several approaches to deal with this issue. The first approach simply re-transforms the changed source model to produce a new version of the target model for each successive update made to the source model. The second approach, usually referred to as incremental transformation, ensures that subsequent changes made to the source model cause appropriate updates on the target model. The second approach provides a more direct solution in terms of identifying the necessary changes to outputs in response to changes to inputs, as opposed to finding the actual outputs themselves. In addition, the second approach is much more efficient, especially in an incremental development environment where small changes frequently occur and constant synchronization is needed [Hearnden et al., 2006]. Both of these approaches, however, assume that the target model was not changed when it is updated. As a result, when following these approaches any modifications that may be made to the target model will be lost the next time a model transformation takes place. There are several approaches to deal with this issue, i.e. preserving any manual updates to the target model. The model merging approach requires re-running the entire transformation, and generating new target models that must then be merged with the previous target models. Other transformations record traceability links between their source and target model elements and use them to perform an incremental update mechanism [Ivkovic and Kontogiannis, 2004; Tratt, 2008].

2.2 Agent-based computing

In the previous section, we have provided some insight into the area of software maintenance and evolution. In this section, we introduce the basic ideas of intelligent agents at a fairly high level of abstraction (section 2.2.1). We also briefly describe the well-known Belief-Desire-Intention (BDI) agent architecture that has been widely used in practice (section 2.2.2). We give an example of a typical BDI execution cycle to show the flexibility offered by the BDI model. Some of the BDI properties are also used in our change propagation framework (see chapter 3). Finally, in section 2.2.3, we highlight the trend of software engineering paradigms and the current position of agent-oriented software engineering (AOSE). In particular, we briefly explain the major challenges that AOSE is facing and how the contribution of our work fits in this picture.

2.2.1 Intelligent agents

The concepts associated with agents, also called *software agents* or *intelligent agents*, have been discussed for many years within the Artificial Intelligence community. There has been a number of different perspectives on agents [Bradshaw, 1997; Franklin and Graesser, 1996]. However, a current consensus considers an agent as "*an encapsulated computer system, situated in some environment, and capable of flexible autonomous action in that environment in order to meet its design objectives*" [Wooldridge and Jennings, 1995]. There are various examples of software agents, for example robot soccer playing at the RoboCup[9] or intelligent shopping agents helping travellers find airfares or holiday bargains. The applications of agent technologies are spread out across different domains, including air traffic control [Ljungberg and Lucas, 1992], business process management [Burmeister et al., 2008], space exploration [Muscettola et al., 1998], manufacturing [Monostori et al., 2006; Shen and Norrie, 1999], information management, and e-commerce [Jennings and Wooldridge, 1998; Munroe et al., 2006].

Agent definitions are often classified into two categories: *strong agency* and *weak agency* [Ferber, 1999; Franklin and Graesser, 1996; Nwana, 1995; Wooldridge and Jennings, 1995]. Agents by weak agency definitions should possess the following characteristics [Wooldridge and Jennings, 1995]:

[9] http://www.robocup.org

- *Situatedness:* Agents are embedded in an environment in terms of using their sensors to perceive the environment and using their effectors to affect the environment. For instance, a robot soccer player can be designed as an agent, which is *situated* in a soccer field. One of the robot's sensors is a group of cameras that keep track of where the ball and other players are. The robot agent also has several effectors such as its legs or its body, which are used to kick or pass the ball.

- *Autonomy:* Agents are able to operate independently, i.e. decide which action they should take, independent of humans or other agents. As a result, agents cannot be directly invoked like objects [Odell, 2002]. In our robot soccer player example, when a robot agent has the ball, the decision whether to kick the ball for a goal or to pass the ball to its teammates is totally up to the agent. This is an example of the autonomy of agents due to the fact that those decisions are made without direct intervention of humans or other robot soccer agents on the field.

- *Reactivity:* Agents can perceive their environment and respond in a timely fashion to changes that occur in it. For example, when our robot soccer player detects the ball being within its control area, it has to quickly perform some actions (to respond to that event) such as passing or shooting the ball.

- *Pro-activeness:* Agents are pro-active if they have goals that they pursue over time. In our example, the major goal of a robot soccer player is to win the game, which can be achieved by scoring goals and defending against conceding goals. The agents pursue this goal by performing actions (e.g. passing the ball to their other teammates, kicking for goals, etc.) that contribute toward accomplishment of the goal.

- *Social ability:* Agents can interact with other agents and humans with the aim of accomplishing their goals. In our example, social ability is reflected by the fact that each robot soccer agent should be able to communicate and coordinate with their teammates, or their coaches (which may be humans).

Strong agency takes a similar perspective in defining which properties an agent should possess, but also views agents as having mentalistic notions such as knowledge, belief, intention and obligation [Shoham, 1993]. According to Dennett [1987], such strong agency (also referred to as an intentional system) can be best described by the intentional stance, which

is a strategy of understanding the behaviour of a system by ascribing mental attitudes such as beliefs, wants, and desires to it. For example, by taking the intentional stance we can predict that a (human) agent will leave the cinema and drive to the restaurant because she sees that the movie is over and is hungry. Shoham also argues that the intentional stance provides a convenient and familiar way of describing, explaining and predicting the behaviour of complex systems (e.g. BDI formalisms of [Rao and Georgeff, 1995] discussed in the next section). Generally, strong agents are built with more artificial intelligence/knowledge-based technologies and to some extent are specified formally [Wooldridge and Jennings, 1995].

2.2.2 The Belief-Desire-Intention (BDI) model

Since the 1980s, the field of agent technology has attracted a substantial amount of interest from researchers [Jennings et al., 1998; Luck et al., 2005]. In particular, there have been a number of different agent theories, architectures, and languages proposed in the literature. One of the most well-established and widely-used agent models is the *Belief-Desire-Intention* (BDI) model. The BDI family of agent theories, languages and systems are inspired by the philosophical work of Bratman [1987] about how humans do resource bounded practical reasoning, i.e. figure out what to do and how to act under limited resource capacity. The key concepts in the BDI model are:

- Beliefs: represent information about the environment, the agent itself, or other agents, from the agent's perspective in terms of the agent's internal and external (environmental) state.
- Desires: represent the objectives to be accomplished in terms of states of the world that the agent wants to reach.
- Intentions: represent the currently chosen courses of action (i.e. *plans*[10]) to pursue a certain desire that the agent has committed to pursuing.

One of Bratman's key contributions was to argue that intentions play a significant and distinct part in the practical reasoning process. He argued that intentions are not reducible to beliefs and desires for several reasons. As agents operate under bounded resources, they must

[10]Plans here are regarded as concrete instances that are being followed, rather than a "recipe" that can be followed (i.e. plan type).

decide on what to pursue (i.e. intentions), rather than waste their resources by continuously weighing between their competing desires. In addition, intentions are used to constrain the choice of plans that the agents considers. Furthermore, in terms of coordinating and planning for future actions, intentions act as a filter on other future options. That means that (a) agents do not adopt courses of actions that clash with the selected intention, and (b) future planning should be based on the assumption that the currently adopted intention will be pursued and (normally) eventually accomplished. For example, if a (human) agent intends to go and watch a soccer match at the stadium, then she should not consider clashing intentions, for example going to a party at the same time. If the party takes place before the match and she desires to attend the party, then the agent can formulate plans to get to the party on her way to the stadium.

The BDI model has been the basis for a range of agent architectures. For example, the classical Intelligent Resource-bounded Machine Architecture (IRMA) proposed by Bratman et al. [1988] has explicit representations of beliefs, desires, and intentions and prescribes how an agent uses them to selects its course of action. The architecture consists of several modules including an intention structure, which is basically a library of partial and hierarchical plans, a means-ends reasoner, an opportunity analyser, a filtering process, and a deliberation procedure. The opportunity analyser is responsible for monitoring the environment in order to determine options for actions. Further options are identified by the means-ends reasoner from the agent's plan library. The filtering process is responsible for determining which of the available options are consistent with the agent's current intentions. However, IRMA also provides an "over-riding" facility where an intention can be adopted in some situations even if it does conflict with existing intentions. Finally, options that successfully pass the filtering process are examined by the deliberation process in terms of making choice between competing options and adopting a new intention.

The IRMA architecture is, however, still at high level of abstraction and is not useful as a practical system. As a result, since the late 1980s a range of work started developing a practical BDI architecture on the basis of Bratman's philosophical foundations. Georgeff and Lansky [1987] developed the Procedural Reasoning System (PRS), which includes a plan library, as well as explicit symbolic representations of beliefs, desires, and intentions. In the early 1990s, Rao and Georgeff proposed an abstract system architecture containing an abstract BDI interpreter (also referred to as an execution cycle) that demonstrates how the

BDI constructs can be used within a computational environment [Rao and Georgeff, 1992; 1995]. They also developed a logical theory that provides a formal framework for defining the concepts of beliefs, desires, and intentions [Rao and Georgeff, 1991]. However, in fact the logical theory deals not with desires, but with goals, which are assumed to be a consistent subset of desires.

At the implementation level, those initial ideas of the BDI model have been modified to suit a practical computational environment. In fact, while BDI theories focus on desires and goals, BDI implementations deal with *events*. Events are significant occurrences that the agent should respond to in some way. Most BDI agent implementation platforms model the change associated with the adoption of new (sub)goals as events. Furthermore, in BDI implementations intentions are viewed as the *plans* which are currently being executed by the agent. Plans are the central concept in the implementations: BDI agents have a collection of pre-defined plan recipes (or types), usually referred to as a plan library. Each plan consists of:

- an invocation condition which defines the event that triggers this plan (i.e. the event that the plan is *relevant* for);
- a context condition (usually referring to the agent's beliefs) which defines the situation in which the plan is *applicable*, i.e. it is sensible to use the plan in a particular situation; and
- a plan body containing a sequence of primitive actions and subgoals that are performed for plan execution to be successful[11]. Subgoals can trigger further plans.

The BDI architecture is realised in a number of agent platforms such as JACK [Busetta et al., 2000], Jadex [Pokahr et al., 2005], PRS [Ingrand et al., 1992], dMARS [d'Inverno et al., 1998], or 3APL [Hindriks et al., 1999] (refer to [Bordini et al., 2006; 2005] for more agent platforms). We describe here a typical execution cycle that implements the decision-making of an agent following an implementation of the BDI architecture. The cycle can be viewed as consisting of the following steps, shown in figure 2.2:

1. An event is received from the environment, or is generated internally by belief changes or plan execution. The agent responds to this event by selecting from its plan library

[11] Generally — and in many actual agent platforms — plan bodies are not just sequences of primitive actions and subgoals, but can include a range of programming constructs, e.g. in JACK [Busetta et al., 2000] plan bodies are a superset of Java.

a set of plans (P_1 - P_k) that are relevant (i.e. match the invocation condition) for handling the event (by looking at the plans' definition).

2. The agent then determines the subset of the relevant plans (P_m - P_n) that is applicable in terms of handling the particular event. The determination of a plan's applicability involves checking whether the plan's context condition holds in the current situation.

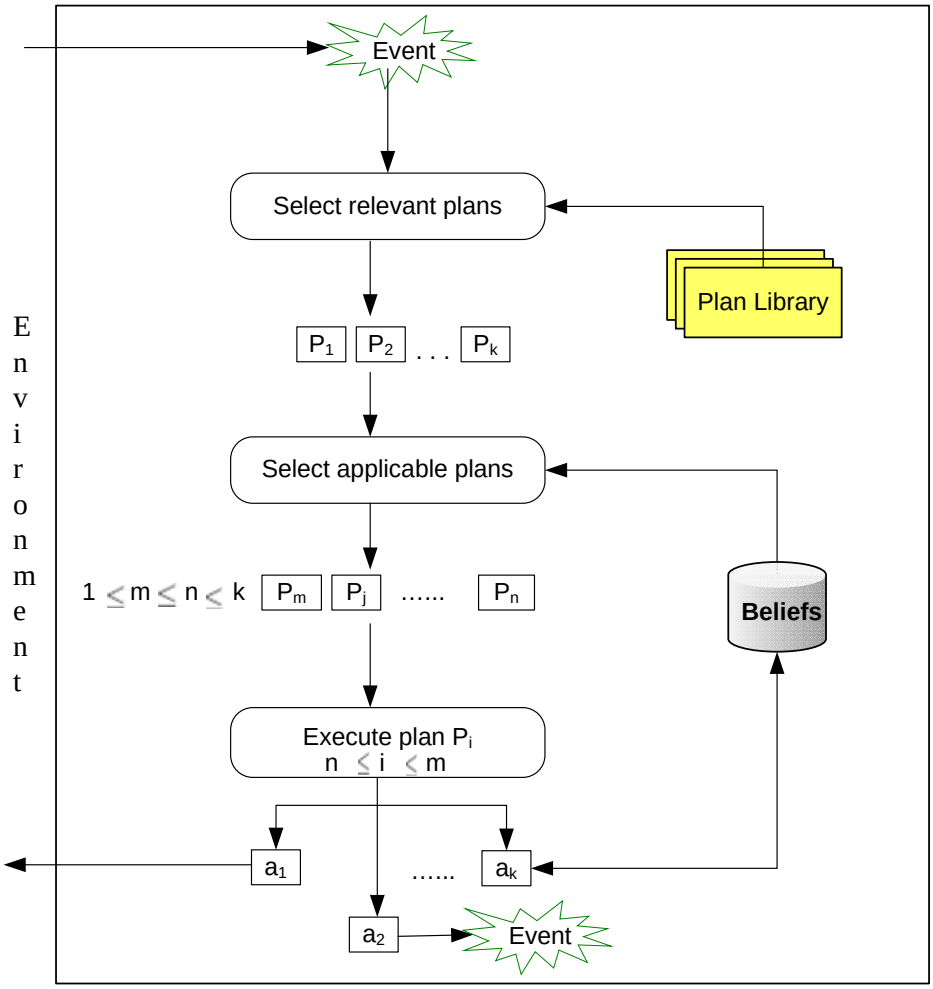


Figure 2.2: A typical BDI execution cycle

3. The agent selects one of the applicable plans (e.g. P_i). This may be based on certain pre-determined priority (e.g. the first applicable plan is selected in JACK's default mechanism) although more sophisticated mechanisms can also be used, depending on the implementation.

4. The agent then executes the selected plan (e.g. P_i) by performing its actions and sub-goals (a_1 - a_k). The actions can be modifying or querying the agent's beliefs, raising new events, or interacting with the environment.

A plan can be successfully executed, in which case the (sub-)goal is regarded to have been accomplished. Execution of a plan, however, can fail in some situations, e.g. a sub-goal may have no applicable plans, or an action can fail, or a test can be false. In these cases, if the agent is attempting to achieve a goal, a mechanism that handles failure is used. Typically, the agent tries an alternative applicable plan for responding to the triggering event of the failed plan. It is also noted that failures propagate upwards through the event-plan tree: if a plan fails its parent event is re-posted; if this fails then the parent of the event fails and so on.

With the above characteristics, BDI agents offer two important qualities: robustness and flexibility. BDI agents are robust since they are able to pursue persistent goals over time (i.e. pro-activeness). In other words, agents will keep on trying to achieve a goal despite previously failed attempts. In order to be able to recover from such failures, agents have multiple ways of dealing with a given goal and such alternatives can be used in case any of them fail. This gives agents flexibility in terms of exercising choice over their actions. Flexibility and robustness are considered as useful qualities that a software system should possess, especially if it operates in complex, dynamic, open and failure-prone environments. With the emergence of distributed information systems, these types of environments are becoming more and more common. Therefore, it is increasingly required to have technologies that are able to cope well in such challenging environments.

Those properties of the BDI model also offer a suitable solution to change propagation. Firstly, BDI agents operate in an event-triggered manner, where events trigger plans, which in turn can create new events resulting in further plans being triggered. This hierarchical relationship between plans is very suitable for representing the cascading nature of change propagation and inconsistency repairing (where fixing an inconsistency by performing an action can cause further inconsistencies requiring further action). In addition, an event can have multiple plans that it can trigger, with plan selection being made at run-time. This allows us to represent multiple ways of resolving a given inconsistency as separate plans, with the choice between them corresponding to available traceability information, design heuristics and (possibly) human intervention. The BDI architecture also offers flexibility as

new or alternative ways of resolving an inconsistency can readily be added via additional plans, without changing the previous structure. We exploit those useful properties of the BDI model to develop an agent-based change propagation framework that is described in the next chapter.

2.2.3 Agent-Oriented Software Engineering

In the previous sections we have briefly explained the concepts of agents and described the BDI model, one of the most well-known agent architectures. The focus of this section is to give an overview of Agent-Oriented Software Engineering (AOSE) and to indicate why and how AOSE has the potential of being an efficient and powerful software engineering approach. The key ideas, research questions and current challenges in AOSE are also briefly discussed in this section.

Since the late 1980s, the object-oriented paradigm gained mainstream acceptance as a new solution to develop software and quickly became the dominant software engineering approach until now. There are a number of advantages that are offered by object-orientation such as data abstraction, information hiding, encapsulation, concurrency [Van Vliet, 2001]. However, the object-oriented approach's benefits are not all that they are often claimed by its proponents . In fact, earlier studies have shown that object-oriented programming is relatively difficult to learn and practice [Armstrong, 2006; Morris et al., 1999; Vessey and Conger, 1994]. In addition, although there have been claims that the object-oriented approach reduces the gap between the real world and the software application, it may not be effective as the primary tool for communication between clients and developers at the beginning stages of software development [Glass, 2002]. Furthermore, a recent study has also indicated that while object-oriented concepts (e.g. objects, classes, or inheritance) appeal to the designer's intuitions of the real world, object-oriented design places them in a formal context where those intuitions may be misleading [Hadar and Leron, 2008].

Recently, there have been several new approaches, which extend object-orientation in order to improve the reusability and other aspects of software, such as component-ware, design patterns, and application frameworks. However, there have been several indications that object-orientation may not be able to deal with the increasing complexity of software systems [Jennings and Wooldridge, 1999]. These emerging complexities result from different sources. Firstly, the rapid and radical change of the information system environment, e.g.

the Internet, has introduced a great deal of complexity to software. Software applications have become more interconnected, decentralized and even more interdependent. In addition, the growing number of interactions between subcomponents, an inherent property of large systems, also lifts the complexities within software. As a result, building high-quality and industrial-strength software becomes more and more difficult.

Agent Oriented Software Engineering (AOSE) is a promising new approach to software engineering that uses the notion of agents as the primary method for analysing, designing and implementing software systems [Jennings, 2001]. The effectiveness of AOSE resides in its ability to translate the distinctive features of agents (discussed in section 2.2.1) into useful properties of (complex) software systems and to provide an intuitive metaphor that operates at a higher level of abstraction compared to the object oriented model.

Firstly, the technical embodiment of agency can lead to reduced coupling, resulting in software systems that are more modular, decentralized and changeable. Indeed, the autonomous property of agents can be viewed as encapsulating invocation [Odell, 2002]. Any publicly accessible method of an object can be invoked externally, and once the method is invoked the object performs the corresponding actions. On the other hand, when receiving a message, an agent has control over how it deals with the message. This ability to encapsulate behaviour activation (action choice) is very useful in open environments in which the system consists of organisations that have different goals [Jennings and Wooldridge, 1999]. Additionally, the robustness, reactiveness and pro-activeness also results in reduced coupling [Padgham and Winikoff, 2004]. Once an agent acquires a goal, it commits to achieve the goal by possibly trying different alternatives in responding to changes in the environment. This means that there is no need for continuous supervision and checking since the agent solely takes responsibility for accomplishing its adopted goals. As a result, it leads to less communication and thus reduced coupling.

Loose coupling and strong encapsulation brought by agents are important, especially because they facilitate the process of evolving software by localising changes to an agent or a group of agents. For instance, the BDI architecture (discussed in the previous section) can be used to model and implement goal-directed process selection [Georgeff, 2006]. Traditionally, the calling process contains the names of the called processes (and possibly other information such as the locations, the data needs, or even the implementation), and the conditions specifying which (process) to call in which circumstance. The major disadvantage of

this conventional approach is that the calling process is dependent on the called processes, and thus they are not able to be developed independently of one another. A goal-directed approach can separate the conditions of use from the calling processes and place them in the called processes. As a result, processes become loosely coupled and process selection is made dynamically at run time based on the usage context. In addition, if any chosen process fails, the call is made again (i.e. reposted) and a new matching process is invoked. This offers a better and more automatic handling of exceptions or failures. Furthermore, called processes can be created or changed without affecting the existing ones and the calling process. These benefits multiply each time the called process is reused in other calling processes.

It has also been argued that AOSE, equipped with the rich representation capabilities of agents, is suitable (and reliable) for modelling complex organisational processes [Jennings et al., 1998; Luck et al., 2005; Munroe et al., 2006]. Jenning and Wooldridge in [Jennings, 2001; Jennings and Wooldridge, 1999] have shown that agent-orientation facilitates complexity management in three aspects: decomposition, abstraction, and organisation. Firstly, they argue that decomposing the problem space of a complex system in an agent-oriented way is very effective. Secondly, they are able to demonstrate the ability of agents in representing high-level abstractions of active entities in a software system, and consequently reducing the gap between business users and system architects. Finally, they explain why it is appropriate to apply an agent-oriented philosophy to the modelling and managing of organisational relationships in such a way that the dependencies and interactions in those complex organisations are effectively dealt with.

However, ultimately, the proof of AOSE is its application in the real world. There are many applications (e.g. [Burmeister et al., 2008; Monostori et al., 2006; Munroe et al., 2006]) and some of these are showing significant measurable benefits [Benfield et al., 2006].

As agents have been increasingly recognised as possibly the next prominent paradigm of developing software, there has been a growth of interest in agent-oriented software engineering. A significant amount of AOSE work has focussed on developing new methodologies and tools for software development using the agent concepts. In fact, as far as we are aware, there have been nearly fifty agent-oriented methodologies proposed to date. Those methodologies (e.g. Gaia [Zambonelli et al., 2003], Tropos [Bresciani et al., 2004], Prometheus [Padgham and Winikoff, 2004], O-MaSE [DeLoach, 2005], PASSI [Cossentino, 2005] etc.) offer notations and models, methods and techniques, processes and (for some methodologies) tool support

that a software developer can use to develop an agent-based application [Bergenti et al., 2004; Henderson-Sellers and Giorgini, 2005].

In the past few years, there has been also a range of work on comparison of the existing agent-oriented methodologies, e.g. [Al-Hashel et al., 2007; Dam and Winikoff, 2003; Elamy and Far, 2006; Sturm and Shehory, 2003; Tran and Low, 2005]. The results from such studies have shown that there are a range of similarities between different methodologies. For instance, they often provide reasonable support for basic agent-oriented concepts such as autonomy, mental attitudes, pro-activeness, and reactiveness. However, each of them also has its own particular strengths and weaknesses. As a result, there have been suggestions concerning methodological integration in a similar way as the Unified Modelling Language (UML) has become standard in the context of object oriented approaches. In fact, there has been work on developing a common metamodel that can be widely accepted by the AOSE community [Bernon et al., 2006] or on unifying graphical notation for several prominent agent-oriented methodologies [Padgham et al., 2008], and there is current work on agent standardisation by FIPA/OMG[12].

Despite its popularity and attractiveness as a research area, agent technology generally and AOSE particularly still face many obstacles in being widely adopted by the industry. Many challenges have been identified and discussed in the community in the past few years (e.g. [Zambonelli and Omicini, 2004], and the recent FOSE-MAS session at the AAMAS 2008 conference[13]). There is an increasing demand for better techniques and tools to test, verify and validate agent-based systems during the development cycle. Additionally, AOSE methodologies, techniques and tools should aim to improve not only the development of agent-based software but also its maintenance and evolution [Cuesta et al., 2007].

In this context, our work also aims to make a contribution to the improvement of AOSE in terms of addressing the issue of software maintenance and evolution, an area that has not seen much work in an AOSE context. We have applied our work to an agent-oriented methodology which is extended to support change propagation during the process of maintaining software systems (refer to chapter 4). The methodology that we chose to use is *Prometheus*, a prominent agent-oriented software engineering methodology which has been used and developed more than 10 years. The methodology is described in considerable detail,

[12] http://www.omg.org/cgi-bin/doc?ad/2008-06-02

[13] Future of Software Engineering and Multi-Agent Systems (FOSE-MAS): http://www.cs.kuleuven.be/~danny/fose-mas.php

and has tool support[14]. A detailed description of the methodology can be found in chapter 4.

2.3 Object Constraint Language

The Object Constraint Language (OCL) [Object Management Group, 2006] is a declarative language for describing expressions and constraints on UML models. It has become[15] a subset of the industry standard Unified Modelling Language (UML) [Object Management Group, 2005] and is one of the core technologies underpinning the recently emerging Model Driven Architecture[16] paradigm [Kleppe et al., 2003]. OCL supplements UML by expressing constraints that cannot be described (or are difficult to describe) using only UML. In addition, OCL offers a proper solution to formalising UML in terms of having neither the ambiguities of natural language (due to freedom of interpretation) nor the inherent difficulty of using complex mathematics. As a result, OCL becomes useful since it allows a developer to create a highly specific set of constraints that govern many essential aspects of the business model of a system. In particular, OCL is an integral part of MDA development since precise (consistent and coherent) models can be built using the combination of UML and OCL [Warmer and Kleppe, 2003].

OCL is deeply integrated with UML in that every OCL expression depends on the types (i.e. the classes, interfaces, etc.) that are defined in the UML diagrams. Figure 2.4 shows a simplified excerpt of the OCL grammar. There are generally four types of constraints that can be expressed using OCL: an invariant , a pre-condition, a post condition, and a guard. A pre-condition or post-condition to an operation is a condition that must hold at the moment just before or after its execution (i.e. *OperationContextDecl*). Meanwhile, a guard is a constraint (specifically referring to statechart diagrams) which describes a condition that must hold before a state transition fires. We are, however, most interested in *invariants* which specify conditions that must hold in all instances of the class, type, or interface in a UML diagram (i.e. *Invariant*). In this sense, all invariants are written using OCL expressions in such a way that the expressions evaluate to true if the invariants hold. In our framework, a consistency constraint contains an OCL expression, specifying an invariant for the associated metamodel elements.

[14] http://www.cs.rmit.edu.au/agents/pdt
[15] The latest version is OCL 2.0.
[16] http://www.omg.org/mda

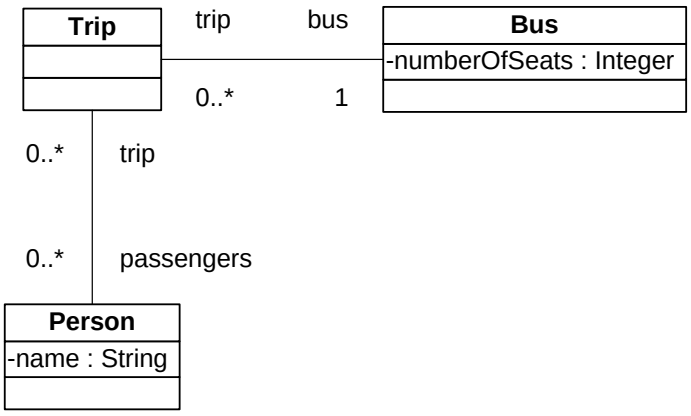


Figure 2.3: An example UML class diagram

We now briefly explain the main parts of the OCL syntax that are used in this thesis by giving examples of how OCL can be used to specify an invariant constraint. For more detail on OCL including all the syntax, grammar, and other specifications, we refer the readers to [Object Management Group, 2006]. Let us consider the UML model shown in figure 2.3, describing three classes: Trip, Person, and Bus. An association between class Trip and class Bus indicates the single bus (note the cardinality 1 on the side of the Bus class) that is used for the trip. Meanwhile, an association between class Trip and class Person indicates that a particular group of persons are the passengers on the trip. The multiplicity many (0..*) on the side of the Person class implies that there is no limit on the number of passengers participating the trip. However, in practice the number of passengers on a trip must be not greater than the number of seats on the bus that is used in the trip. This restriction cannot be expressed in the UML diagram. Rather, we use the following OCL constraint:

context *Trip*
inv : *self.passengers→size() <= self.bus.numberOfSeats*

In the above constraint, the *context* keyword refers to the model entity for which the OCL expression is defined. The model entity is usually a class, interface, datatype or even component, and is generally termed as a *Classifer* in the UML standard. Each OCL expression is written in the context of an instance of a specific type. For instance, in the above constraint the *Trip* class is the *contextual type* of the OCL expression. Meanwhile, the keyword *self* is used to refer to the *contextual instance*, which is the instance of *Trip* in our example. The

ContextDeclaration	::=	*ClassifierContextDecl* | *OperationContextDecl*
ClassifierContextDecl	::=	"*context*" *pathName Invariant*
Invariant	::=	"*inv*" (*simpleName*)? : *OclExp Invariant*
OclExp	::=	*LetExp* | *LogicalExp*
LogicalExp	::=	*RelationalExp* (*LogicalOperator RelationalExp*)∗
RelationalExp	::=	*AdditiveExp* (*RelationalOperator AdditiveExp*)?
AdditiveExp	::=	*MultiplicativeExp* (*AddOperator MultiplicativeExp*)∗
MultiplicativeExp	::=	*UnaryExp*(*MultiplyOperator UnaryExp*)∗
UnaryExp	::=	(*UnaryOperator PostfixExp*) | *PostfixExp*
PostfixExp	::=	*PrimaryExp* | ((“.” | “− > ”) *ModelPropertyCallExp*)∗
PrimaryExp	::=	*LiteralExp* | *ModelPropertyCallExp* | *IfExp*
ModelPropertyCallExp	::=	*OperationCallExp* | *AttributeCallExp*
		NavigationCallExp | *LoopExp*
NavigationCallExp	::=	*AssociationEndCallExp* | *AssociationClassCallExp*
LoopExp	::=	*IteratorExp* | *IterateExp*
OperationCallExp	::=	*OclExp simpleName OclExp* |
		OclExp “ → ” *simpleName* “(” *arguments*? “)”
		OclExp “.” *simpleName* “(” *arguments*? “)”
		simpleName “(” *arguments*? “)”
		pathName “(” *arguments*? “)”
		simpleName OclExp
AttributeCallExp	::=	*OclExp* “.” *simpleName* | *simpleName* | *pathName*
IfExp	::=	“*if*” *OclExp* “*then*” *OclExp*
		“*else*” *OclExp* “*endif*”
IteratorExp	::=	*OclExp* “ → ” *Iterator* “(” *VariablesDecl* “ | ” *OclExp* “)”
IterateExp	::=	*OclExp* “ → ” “*iterate*” “(” *VariablesDecl* “ | ” *OclExp* “)”
Iterator	::=	“*forAll*” | “*exists*” | “*select*” | “*reject*” | “*one*” |
		“*collect*” | “*any*” | “*isUnique*” | “*sortedBy*”
LiteralExp	::=	*EnumLiteral* | *CollectionLiteral* | *PrimitiveLiteral*
LogicalOperator	::=	“*and*” | “*or*” | “*xor*” | “*implies*”
RelationalOperator	::=	“ = ” | “ <> ” | “ < ” | “ > ” | “ <= ” | “ >= ”
UnaryOperator	::=	“ − ” | “*not*”
AddOperator	::=	“ + ” | “ − ”
MultiplyOperator	::=	“ × ” | “ : ”

Figure 2.4: An excerpt of the OCL grammar (adopted from [Object Management Group, 2006] and [Object Management Group, 2005])

invariant is expressed in the second line of the constraint declaration (although it is legitimate to put the whole expression in just one line) following the keyword *inv* (referring to an invariant instead of other types of constraint) and a colon.

OCL is not only an expression language but also a navigation language for graph-based models like UML models. It means that starting from a specific instance, we can navigate an association (on the class diagram) to refer to other instances and their properties (i.e. *NavigationCallExp*). Navigation can be done using the association ends, which is written as *instance.rolename*. Depending on the cardinality of the association end *rolename*, the value of this expression can be a single instance or a set of instances. For example, in the above constraint the *self.passengers* expression, going from *self* (an instance of Trip) to class Person (which has an association to class Trip with role name *passengers*), results in a Set of Persons because of the multiplicity 0..* on the side of the Person class. On the other hand, *self.bus* refers to a single instance of Bus because the multiplicity of the association (between classes Trip and Bus) on the side of class Bus is one.

In OCL, we can also refer to the value of an attribute (i.e. *AttributeCallExp*). For example, in our model the Bus class has an attribute *numberOfSeats*. As a result, since *self.bus* is an instance of Bus, *self.bus.numberOfSeats* is the value of the attribute *numberOfSeats* of this instance. Informally, *self.bus.numberOfSeats* means the number of seats on the bus that is used for a given trip *self*.

A navigation over associations can result in not only Sets but also other OCL pre-defined collection types such as Bags (resulting from combined multiple navigations) and Sequences (resulting from navigation over associations adorned with "ordered"). However, we only focus on Sets here since it is the main collection type that we deal with. OCL supports a large number of pre-defined operations that can be used to apply a property on a Set. A property of the set itself is accessed using an arrow "→" followed by the name of the association operation. For example, *self.passengers→size()* refers to the *size* property of the Set *self.passengers*, which results in the number of passengers on the Trip *self*. In table 2.1, we summarise the set operations that are supported in OCL[17]. This includes all predefined collection operations that use an iterator (*LoopExp*) such as *iterate*[18], *select*, *collect*, *reject*, *forAll*, *exists*, and other pre-defined set operations such as *union*, *intersection*, etc. (which

[17] *SE* is denoted as a set.

[18] The *iterate* operation is the most fundamental and complex loop operations in OCL. More details of this operation can be found in [Object Management Group, 2006]. Although we do not directly deal with this operation, we address other simple forms of loop operations such as *forAll*, *exist*, *select*, etc.

are part of *OperationCallExp*[19]).

OCL also supports logical connectives including *and*, *or*, *xor*, *not*, *implies* (i.e. *LogicalExp*), and *if − then − else* (i.e. *IfExp*). For example, the constraint presented earlier can be extended to include the condition that passengers who take part in the trip must not have the same name. We use the *and* operator to connect the first constraint with the newer one, resulting in the following OCL expression:

context *Trip*
inv : *self.passengers→size() <= self.bus.numberOfSeats* and *self.passengers→isUnique(name)*

OCL is a typed language and the pre-defined basic types in OCL are Boolean, Integer, Real, and String. All classifiers from the UML model (e.g. classes, interfaces, datatypes, etc.) are types in OCL expressions that are associated to the model. In addition, enumeration types are supported. OCL allows for reusing variables or operations in terms of the *let* expression (i.e. *LetExp*), which enables us to define a variable or operation that can be used instead of a sub-expression. There are several pre-defined operations that apply to all instances in the associated UML model. For example, for dealing with the (direct or indirect) type of an instance, we can use *oclIsTypeOf*, *oclIsKindOf*, or *oclAsType* operations.

As OCL use has become more widespread, there have been a number of tools supporting OCL available in both academia and industry [Álvarez et al., 2003]. OCL tools aim to support and handle OCL expressions to different extents. The common functionalities of those tools such as Octopus[20], Dresden OCL2 Toolkit[21], Naomi[22], OCL Library[23], OCLE[24], and USE[25] are parsing/compiling OCL expressions and evaluating OCL expressions against UML models. Several tools such as Dresden OCL2 Toolkit, Octopus, OCL Library, etc. also offer the capability of generating Java and even SQL (e.g. Dresden OCL2 Toolkit) from UML models with OCL expressions. There are stand-alone tools (e.g. Dresden OCL Toolkit) or tools that are integrated with a modelling tool (e.g. Octopus is an Eclipse plugin, and

[19] An operation call can have several forms, such as a pre-defined collection operation, class operation call and so on.
[20] http://www.klasse.nl/octopus
[21] http://dresden-ocl.sourceforge.net
[22] https://sourceforge.net/projects/mocl/
[23] http://www.cs.kent.ac.uk/projects/ocl
[24] http://lci.cs.ubbcluj.ro/ocle
[25] http://i12www.ira.uka.de/~key

Operation	**Description**
SE→size()	The number of elements in *SE*.
SE→includes(x)	True if x is an element of *SE*, false otherwise.
SE→excludes(x)	True if x is not an element of *SE*, false otherwise.
SE→includesAll(SE')	True if *SE* contains all the elements of SE'.
SE→excludesAll(SE')	True if *SE* contains none of the elements of SE'.
SE→isEmpty()	True if *SE* contains no elements.
SE→notEmpty()	True if *SE* contains one or more elements.
SE→sum()	The addition of all elements in *SE*. Elements must be of a type supporting the + operation, e.g. Integer and Real.
SE→exist(c)	True if there is at least one element in *SE* for which constraint c is true.
SE→forAll(c)	True if constraint c is true for all elements in *SE*.
SE→isUnique(expr)	True if the expression *expr* evaluates to a different value for each element in *SE*.
SE→sortedBy(expr)	The Sequence consisting of all elements of *SE* in which the element for which the expression *expr* has the lowest value comes first, and so on. The type of the *expr* expression must be comparable.
SE→iterate(expr)	A value obtained by iterating over all elements in *SE*.
SE→one(c)	True if there is exactly one element in *SE* for which constraint c is true.
SE→any(c)	Any element in *SE* for which constraint c is true.
SE1→union(SE2)	The union of *SE*1 and *SE*2.
SE1 = SE2	True *SE*1 and *SE*2 contain the same elements.
SE1→intersection(SE2)	The intersection of *SE*1 and *SE*2, i.e. the set of all elements that are in both *SE*1 and *SE*2.
SE1 − SE2	The element of set *SE*1, which are not in set *SE*2.
SE→including(x)	The set containing all elements of *SE* plus x.
SE→excluding(x)	The set containing all elements of *SE* without x.
SE1→symmetricDifference(SE2)	The set containing all the elements that are either in *SE*1 or *SE*2 but not in both.
SE→select(c)	A subset of *SE* containing all elements for which constraint c is true.
SE→reject(c)	A subset of *SE* containing all elements for which constraint c is false.
SE→collect(expr)	The Bag of elements that results from applying *expr* to every member of *SE*.
SE→count(x)	The number of times that element x occurs in *SE*.
SE→asSequence()	The Sequence consisting of all the elements from *SE*, in undefined order.
SE→asBag()	The Bag consisting of all elements from *SE*.

Table 2.1: Set operations supported in OCL

Oclarity[26] is an AddIn for Rational Rose). In fact, commercial visual modelling applications (e.g. Borland Together[27]) have started to provide support for OCL together with the MDA development. Apart from the core functionalities we mentioned earlier, the commercial tools also offer usability features such as syntax highlighting, and OCL expression constructing.

An important aspect that in our view is missing from those tools is the capability of dealing with violated OCL constraints. Constraint violations may indicate that the UML model is not well-formed or inconsistent. Therefore, resolving those violations is equally important as identifying them. In order to deal with this issue, it is necessary to provide a (semi)-automated mechanism for working out the resolutions for particular OCL constraints and executing these resolutions (i.e. making changes to the model). This is also one of the issues that this dissertation aims to address.

2.4 Chapter summary

In this chapter, we have given an overview of software maintenance and evolution by briefly reviewing its history, presenting its definitions and key concepts, and emphasizing its significant importance in software development. We have also explained the three major types of maintenance: perfective, adaptive, and corrective, as well as briefly describing other variations of this classical taxonomy. As the aim of our work is to improve change propagation in software maintenance, we give it more attention in this chapter. More specifically, a typical change mini-cycle process has been described to show the role of change propagation in software change and how it fits in with other activities such as program comprehension, change impact analysis, and restructuring. We also reviewed different approaches that have been proposed to deal with the change propagation issue in software maintenance. Those approaches address various aspects of the problem including developing a formalism to model the process of change propagation, managing inconsistencies caused by software changes, and transforming models. We have also discussed the advantages and disadvantages of those approaches in order to highlight the gaps that our work can aim to fill.

The second part of this chapter introduced the agent-based paradigm. It started with a brief description of software agents and their key concepts including situatedness, autonomy, reactivity, pro-activeness, and social activity. We have also discussed the advantages of the

[26] http://www.empowertec.de/products/rational-rose-ocl.htm
[27] http://www.borland.com/us/products/together

notion of agency over other existing entities such as objects. In order to strengthen our points, we briefly described the well-known Belief-Desire-Intention (BDI) agent architecture to show some aspects of its flexibility in an execution cycle. Finally, our description of the agent paradigm ended with an overview of agent-oriented software engineering (AOSE), which is also the context of our work.

The final part of the chapter introduced the Object Constraint Language (OCL), which our work is extensively based on. We have briefly described some of the key OCL constructs and explained how OCL can be used with UML models using a small example. Finally, we gave a brief review of existing tool support for OCL with an indication that there has been a lack of support for resolving OCL constraint violation, which is one of our work's objectives. In the next chapter, we begin describing our work with a description of an agent-based change propagation framework that we have developed.

Chapter 3

Change Propagation Framework

The previous chapter provided some insight into the problem which this research attempts to tackle. We have briefly touched on the concepts related to agents, agent-based systems and agent-oriented software engineering. In addition, we provided a broad overview of the research area of software maintenance and evolution with a special focus on approaches and techniques for planning and implementing changes, and managing inconsistencies in and between models. Furthermore, we also reviewed the Object Constraint Language, a recently well-known standard for UML modelling. On this basis, in this chapter we present our approach to the issue of change propagation in maintaining and evolving agent systems. We first discuss how to develop consistency relationships in design models (section 3.1). In this section we also provide a definition of a model and explain why model consistency is important. In addition, we discuss how consistency relationships can be defined using a metamodel and a set of constraints. We also discuss how such constraints can be extracted from a design model. In section 3.2, we present our approach to change propagation based on fixing inconsistencies in the design. We also explain how our work is situated in the area of model inconsistency management. Furthermore, we provide a classification of repair actions that can resolve inconsistencies. Section 3.3 is the crucial part of this chapter in which we describe an agent-based change propagation framework. We present an architectural view of the framework as well as explain why the well-known Belief-Desire-Intention agent architecture is a natural match for representing and implementing a change propagation mechanism.

3.1 Building consistency relationships in design models

3.1.1 What is a model?

The central subject of our research is a (design) model. Before we discuss how to deal with changes made to a model, we need to understand what a model is in the context of software development. In Mellor et al. [2003], a model is defined as a *"coherent set of formal elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis, such as:*

- *Communication of ideas between people and machines*
- *Completeness checking*
- *Race condition analysis*
- *Test case generation*
- *Viability in terms of indicators such as cost and estimation*
- *Standards*
- *Transformation into an implementation"*

Due to the various benefits mentioned above, models have been widely used to provide an abstract representation of the important aspects of the system under development. In fact, models for representing system designs are in general the core component of any software engineering methodology. Advocates of model-driven engineering even consider models as the primary assets [Mellor et al., 2003].

The term *model* is, however, used in many contexts and usually has a different meaning. For example, a class diagram is also referred to as a class model. An interaction diagram is sometimes called a interaction model. Are these two models not related to each other or should they be regarded as being part of the same thing? In order to answer this question, let us explain what is meant by a model and discuss some of its key properties.

Since our objective is to propagate changes by fixing inconsistencies in design models, we have adopted the following definition of a model from [Kleppe et al., 2003].

Definition 1. *A model describes a system using a well defined (modelling) language, which has clear syntax and semantics suitable for both human and computer interpretation.* ■

This definition makes a distinction between models in terms of the language in which each model is written. For instance, a UML model is written in UML which contains concepts such as classes, methods, objects, etc. On the other hand, a Prometheus model is written in a language that contains concepts specific to agents such as goal, plans, events, etc.

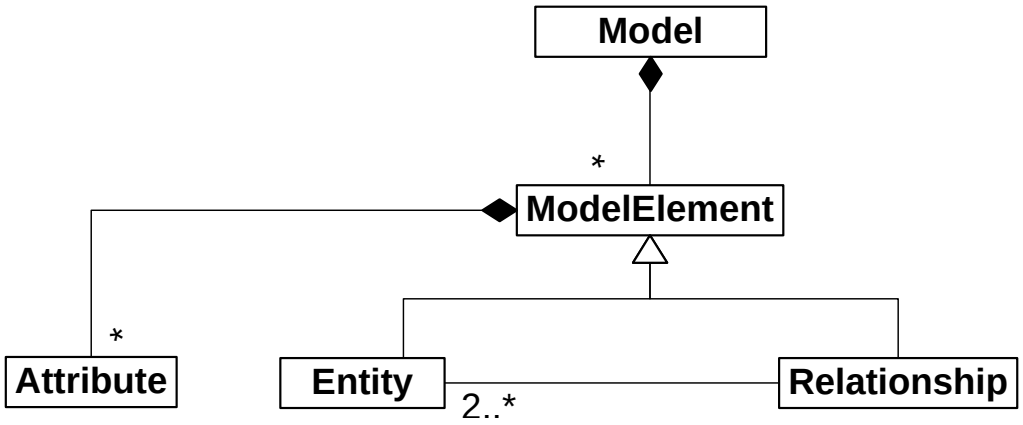


Figure 3.1: A model and its containments

The above definition leads to our notion of a model which is adopted from [Warmer and Kleppe, 2003].

- A model is a consistent and coherent set of model elements (refer to figure 3.1). Model elements can be either model *entities* (e.g. agent, class) or *relationships* between two or more entities (e.g. a triggering relationship between an event and a plan, or an association between classes). Model entities also have *attributes* such as name. We consider attributes that have primitive types (e.g. integer, string) instead of referencing types, which are regarded as relationships. For instance, an agent has a number of goals. Instead of considering a set of goals as an attribute of an agent, we view an agent as having an aggregation relationship with goals.

- A diagram provides a certain view of the model elements in a model. For instance, a UML class diagram depicts a structural view of a model whereas a sequence diagram represents a dynamic view of the same model. Different views of a model are all written in the same language. Figure 3.2 shows how different diagrams in UML are all views on the same model and are all written in the same language (i.e. UML).

  The multi-view approach is widely used in software engineering due to several reasons

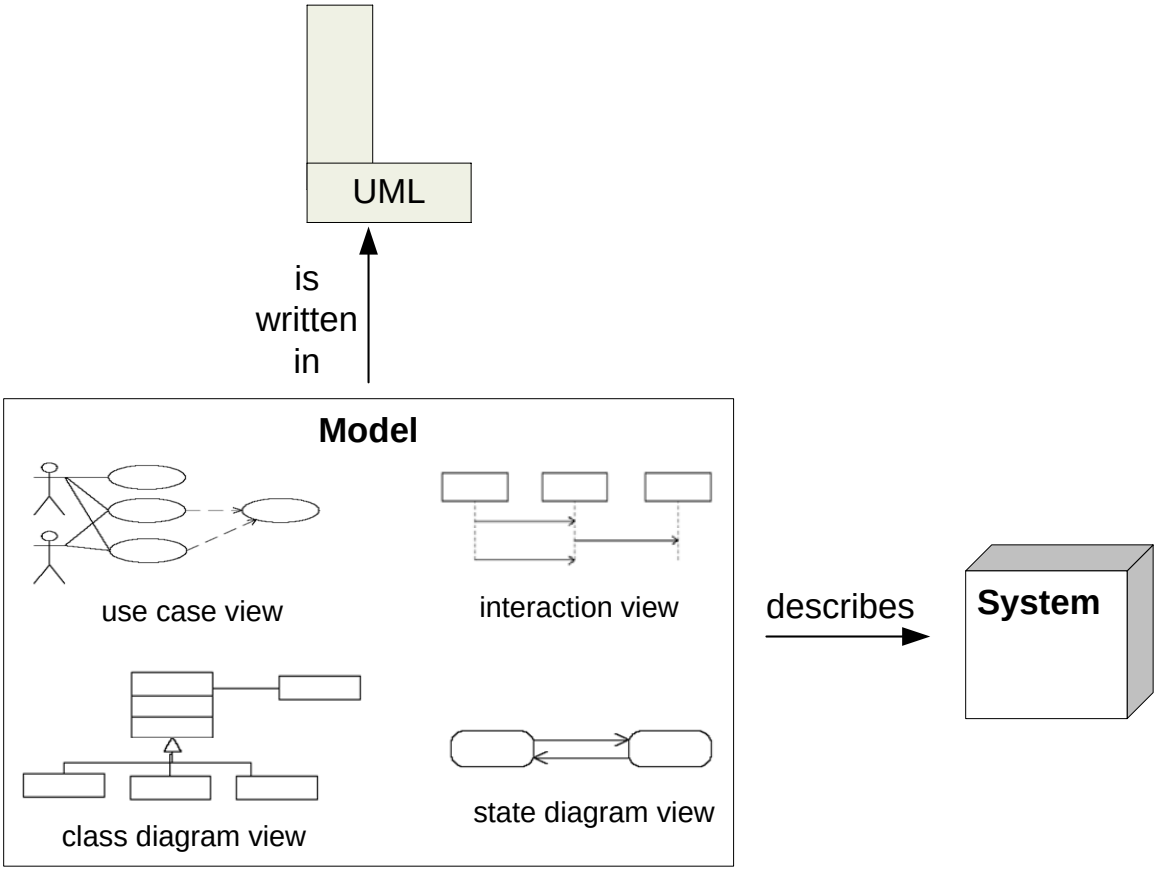


Figure 3.2: Different views of one system in one model (redrawn from [Kleppe et al., 2003])

> [Kruchten, 1995; Object Management Group, 2005]. A complex system usually has many aspects such as how its components are structured, how they interact with each other or how data flows and is processed. In addition, different stakeholders may have different viewpoints of the system, e.g. the customer is more interested in the system's functionalities whereas the focus of a designer might be the internal structure of the system. As a result, no single diagram is sufficient to represent the whole system's model and consequently a range of different views is needed to provide multiple perspectives of the system under development.

Since a model has multiple views, it is difficult to maintain consistency between them. Especially during the maintenance process, the designer usually focuses on making changes to some of the views while he/she may ignore the others. This partial view of the model can cause serious issues later on if the model is made inconsistent.

3.1.2 How to define consistency in models?

We have defined a model as a description of a system using a well-defined modelling language which is suitable for computer interpretation. Metamodelling is a widely used mechanism to define such a language [Atkinson and Kühne, 2003]. A metamodel is also a model which describes the abstract syntax of a modelling language, i.e. a definition of all the concepts and the relationships existing between concepts that can be used in that language. Similarly, a metametamodel defines the language in which metamodels are expressed. In principle, there is an infinite number of layers of metamodelling. In practice however, the four levels of modelling have been generally agreed upon and have become a standard [Miller and Mukerji, 2003]. In this context, the models at the lowest level are called the M0 models and contain instances, i.e. the actual items that are being modelled. The next layer contains M1 models, while meta models are called M2 and meta meta models are called M3. A model is said to conform (or also called well-formed) to its metamodel like a program conforms to the grammar of the programming language in which it is written.

It is very important that each view of a model must be both syntactically and semantically consistent [Spanoudakis and Zisman, 2001]. Syntactic consistency ensures that a model's view conforms to the model's abstract syntax, i.e. a metamodel, which guarantees that the overall model is well-formed. For instance, a well-formed UML model contains classes, operations, messages, and so on because the UML metamodel (see figure 3.4 on page 54) defines what is a class, message, operation and so on. However, such elements (e.g. classes) may not be contained in a valid Prometheus model because they are not defined in the metamodel of Prometheus (see figure 3.3 on page 53). On the other hand, semantic consistency requires different views of a model to be semantically compatible (i.e. coherence). For instance, the message calling direction in a UML sequence diagram must match the class association direction in a class diagram.

Such consistency requirements upon a model are often expressed using its metamodel and a set of constraints that specify conditions that a well-formed and consistent model should satisfy. In this context, we adopt a simple definition of what actually constitutes an inconsistency: *an inconsistency occurs if and only if a (consistency) constraint has been broken.* Such a constraint explicitly describes some form of relationship or fact that is required to hold. Constraints may describe syntactic and semantic relationships between model elements. They may also be used to prescribe coherence relationships between different views of a model,

i.e. intra-model or horizontal consistency as defined in [Spanoudakis and Zisman, 2001]. Our approach is also applicable to constraints that express consistency between different models (or referred as inter-model or vertical consistency by Spanoudakis and Zisman [2001]) provided that those models are defined based on the same metamodel. In addition, constraints can be used to impose best practices or industry standards on designers, e.g. a constraint requiring that all constructors of a class are declared private and that the class provides static creation operations (factory pattern) [Gamma et al., 1995]. Finally, constraints may be used to describe specific requirements related to a particular domain, e.g. there could be a constraint that all agents in the system need to subscribe to a particular agent.

We use the Object Constraint Language [Object Management Group, 2006] to specify constraints. OCL is part of the UML standard which is used to specify invariants, pre-conditions, post-conditions and other kinds of constraints imposed on elements in UML models. Below (constraint 1) is an example of an OCL constraint that defines the semantics of relationships between agents, roles and percepts which are described in the metamodel in figure 3.3[1]. In the OCL notation *"self"* denotes the context node (in this case an Agent) to which the constraints have been attached and an access pattern such as *"self.perceptsEntityReference"* indicates the result of following the association between an agent and a percept (in the metamodel), which is, in this case, a collection of percepts which are handled by the agent. OCL also has operations on collections such as *"SE→includes(x)"* stating that a collection SE must contain an entity x, or *"SE→exists(c)"* specifying that a certain condition c must hold for at least one element of SE, or *"SE→forAll(c)"* specifying that c must hold for all elements of SE. In section 2.3 (on page 39) we provide more detailed information on OCL.

For example, the following constraint (with regard to the metamodel in figure 3.3), which could be expressed in a more traditional form as $\forall\, p \in self.perceptsEntityReference \bullet \exists\, r \in self.rolesEntityReference \bullet p \in r.perceptsEntityReference$, states that: considering the set of percepts that are handled by the agent (*self.perceptsEntityReference*), for each of the percepts (p) if we consider the roles played by that agent (*self.rolesEntityReference*) then one of these roles (r) must include the current percept (p) in the list of percepts that it handles (*r.perceptsEntityReference*).

Constraint 1. *Any percept handled by an agent must be handled by at least one of its roles.*

[1]This figure shows that an agent can play a number of roles and handle some percepts, and that a role can also handle several percepts. The complete Prometheus metamodel and consistency constraints are provided in chapter 4.

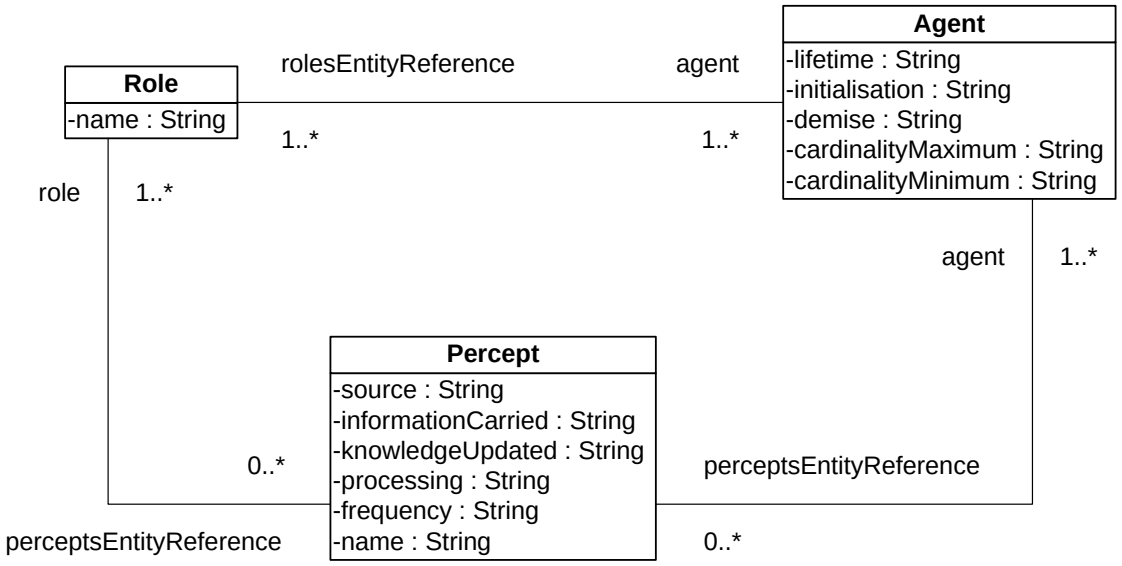


Figure 3.3: Prometheus Metamodel (Excerpt)

Context Agent inv*:*

self.perceptsEntityReference→forAll(p : Percept |

self.rolesEntityReference→exists(r : Role | r.perceptsEntityReference→includes(p)))

The above constraint is applied to the Prometheus methodology. Below is an example of a consistency constraint for UML 1.5 models (with regard to the UML metamodel in figure 3.4[2]).

Constraint 2. *The name of a message (in sequence diagrams) must match an operation in its receiver's class (in class diagrams).*

Context Message inv*:*

self.receiver.base.operation→exists(op : Operation | op.name = self.name)

The metamodel and constraints can be developed by extracting information such as relationships, dependencies, and even best practices from a methodology. For example, a metamodel for UML and a set of well-formedness constraints have been defined in [Object Management Group, 2005]. In chapter 3, we will discuss how we have developed a metamodel and a collection of consistency constraints for the Prometheus methodology [Padgham and Winikoff, 2004].

[2]It is noted that in practice a ClassifierRole typically has one Class base.

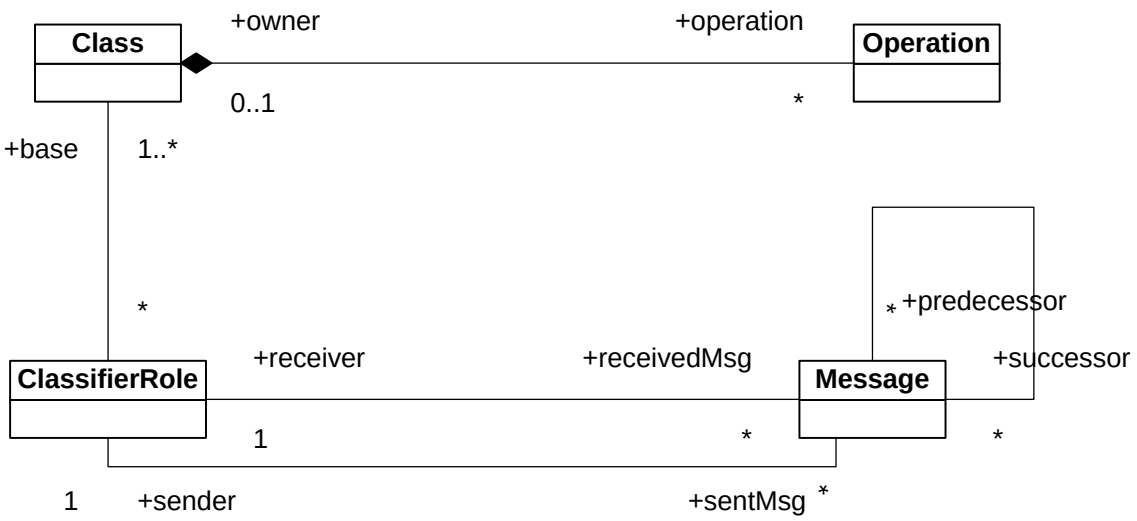


Figure 3.4: UML 1.5 Metamodel (Excerpt)

3.2 An inconsistency based approach to change propagation

When software is modified, typically some primary changes are made and then additional, secondary, changes are made as a result. Change propagation is the process of determining and making these secondary changes. We approach the change propagation issue based on the conjecture that, given a suitable set of consistency constraints, change propagation can be done by fixing inconsistencies in a design. In other words, we propagate changes by finding places in a design where the desired consistency constraints are violated, and fixing them until no inconsistency is left in the design.

There are, however, various strategies to handle inconsistency and to preserve consistency in design models. In this section, we discuss which inconsistency handling strategy we chose to follow. In addition, we look at repair actions that allow for semi-automatic resolution of inconsistencies.

3.2.1 Inconsistency management

Inconsistency management is a process that includes various activities ranging from defining and detecting inconsistencies to diagnosing and handling them [Finkelstein et al., 1996; Grundy et al., 1998; Nuseibeh et al., 2000; Spanoudakis and Zisman, 2001; Van Der Straeten, 2005] . In the previous section, we have discussed how a metamodel and a set of constraints

are developed to specify consistency relationships. In this context, inconsistencies are defined as the occurrence of violation of such constraints. The question remains how does our approach detect, diagnose and handle inconsistencies?

Detection of inconsistencies is the activity of checking for inconsistencies in design models. Our work does not aim to tackle this issue. Rather, we rely on existing techniques and tools for checking inconsistencies. As discussed in chapter 2, there are various approaches for the detection of inconsistencies [Spanoudakis and Zisman, 2001]. In our framework, inconsistencies are in the form of an OCL constraint violation. Therefore, we use existing techniques and tools that check OCL constraints.

Diagnosis of inconsistencies is the activity for identifying the source, the cause and the impact of an inconsistency [Spanoudakis and Zisman, 2001]. In our approach, the source of an inconsistency is a collection of model elements involved in the inconsistency. In addition, we consider the primary changes made to the model as the cause of an inconsistency. It can be useful to analyse the impact of an inconsistency in order to determine the priority with which the inconsistency has to be handled. This task is normally left to the user [Van Der Straeten, 2005] as the impact of an inconsistency depends on the applications and on the current activity of software maintenance and evolution. Therefore, the way that we deal with this issue is by allowing the user to prioritize consistency constraints, e.g. determine which ones are mandatory, and which ones are optional (a more fine-grained priority scheme could also be used).

Inconsistency handling includes activities for identifying methods for dealing with an inconsistency (e.g. repair it, or note it but not fix it, or ignore it), evaluating the impacts and consequences of each of the methods, and determining when to deal with inconsistencies. As also discussed in chapter 2, there are different techniques for handling an inconsistency. A key idea of inconsistency handling is to tolerate inconsistencies [Nuseibeh et al., 2000]. Balzer [1991] was among the first to address this particular issue by proposing a formalism in which inconsistencies can be temporarily marked and resolved later rather forcing resolution at the point of violation. Our approach addresses this issue in several ways. Firstly, when primary changes are made temporary inconsistencies are allowed as in Balzer's approach. The change propagation process is triggered by the user to fix inconsistencies only when he/she decides to do so. Secondly, we allow the user to determine whether an inconsistency is acceptable. This can be done at design time (by determining whether a constraint is

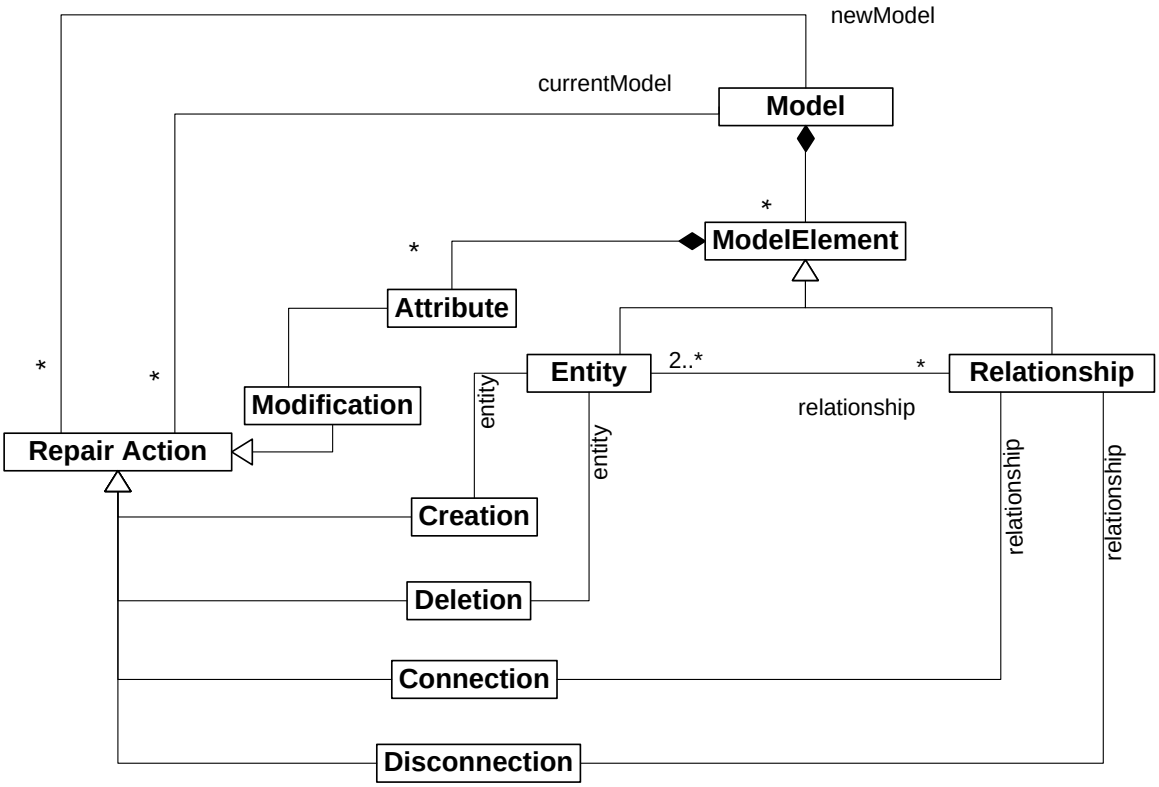


Figure 3.5: A classification of repair actions

included) or at run time (by determining whether a violated constraint instance should be fixed or not). Inconsistencies that should be fixed are resolved by repair actions which modify the design model. We now consider the different kinds of repair actions.

3.2.2 Classification of repair actions

A design model is updated through repair actions performed on the model's elements or their attributes (see figure 3.5). We adopted an evolution model described in [Mens and D'Hondt, 2000] to classify such repair actions into five different primitive types. For each type of repair action we informally describe its effect on the model and also provide a formal OCL representation.

- Creation: a model entity is created, e.g. creating a new agent, which in terms of the (UML) metamodel, implies the instantiation of a metaclass Agent. Creation does not result in any changes to the set of relationships in the current model, and the set of

entities in the new model will be a union of the entities in the current model and the new entity being added.

Context Creation*:*
self.currentModel.entity[3]→*excludes(self.entity)*
self.newModel.entity = self.currentModel.entity→*including(self.entity)*
self.newModel.relationship = self.currentModel.relationship[4]

- Deletion: a model entity is deleted from a model, e.g. an agent is deleted, which implies the deletion of an instance of a metaclass Agent. The entity being deleted should exist in the current model and should not exist in the new model. Moreover, the set of entities in the new model will be the entities in the current model minus the entity being removed. In addition, all the relationships associated with the entity being removed should also be deleted.

  Context Deletion*:*
  self.currentModel.entity→*includes(self.entity)*
  self.newModel.entity = self.currentModel.entity→*excluding(self.entity)*
  self.newModel.relationship = self.currentModel.relationship − self.entity.relationship

- Connection: a relationship is added to a model by connecting two or more entities, e.g. assigning a role to an agent. This action has no effect on the set of entities in the current model. The relationship being added should not exist in the current model. In addition, the set of relationships in the new model will be a union of the relationships in the current model and the relationship being added.

  Context Connection*:*
  self.newModel.entity = self.currentModel.entity
  self.currentModel.relationship→*excludes(self.relationship)*
  self.newModel.relationship = self.currentModel.relationship→*including(self.relationship)*

- Disconnection: a relationship is removed from a model by disconnecting two or more entities, e.g. unassigning a role from an agent. There are no changes to the set of entities in the current model. In addition, the relationship being removed should exist in the current model and should not exist in the new model. Finally, the set of relationships

[3] *self.currentModel.entity* results in a set of entities in the current model.
[4] *self.currentModel.relationship* results in a set of relationships in the current model.

in the new model will be the relationships in the new model minus the relationship being removed.

Context Disconnection*:*
self.newModel.entity = self.currentModel.entity
self.currentModel.relationship→includes(self.relationship)
self.newModel.relationship = self.currentModel.relationship→excluding(self.relationship)

- Modification: an attribute of a model element is modified, e.g. the name of an agent is changed. There should not be any effect on the set of entities and relationships in the current model .

The above taxonomy covers a complete range of fine-grained actions that can be made to a model. In the next section, we discuss a change propagation framework that uses such actions to propagate changes by resolving inconsistencies.

3.3 Architectural overview of our change propagation framework

In this section, we describe the architecture of an agent-based framework to deal with change propagation by fixing inconsistencies in a design model. The framework provides a flexible underlying formalism on which a "change propagation assistant", that offers support to a designer by suggesting additional (secondary) changes once primary changes have been made, can be based.

Figure 3.6 shows an overview of our architecture as a data flow diagram. The key data items we deal with are as follows.

- An **application model**: the current design of a system that is being modified for maintenance and/or evolution purposes. Such a model needs to meet two important criteria. Firstly, it should be in a computer interpretable form so that changes can be automatically made to it, and it can be automatically checked for inconsistencies. For example, the XML Metadata Interchange (XMI) [Object Management Group, 2003] is widely used as an interchange format for UML models, and is supported for importing and exporting by a large number of CASE tools. It is noted that the application's design model can usually be obtained from the CASE tool that is used to develop the design. For instance, ArgoUML[5] supports exporting a model in XMI format.

[5]http://argouml.tigris.org

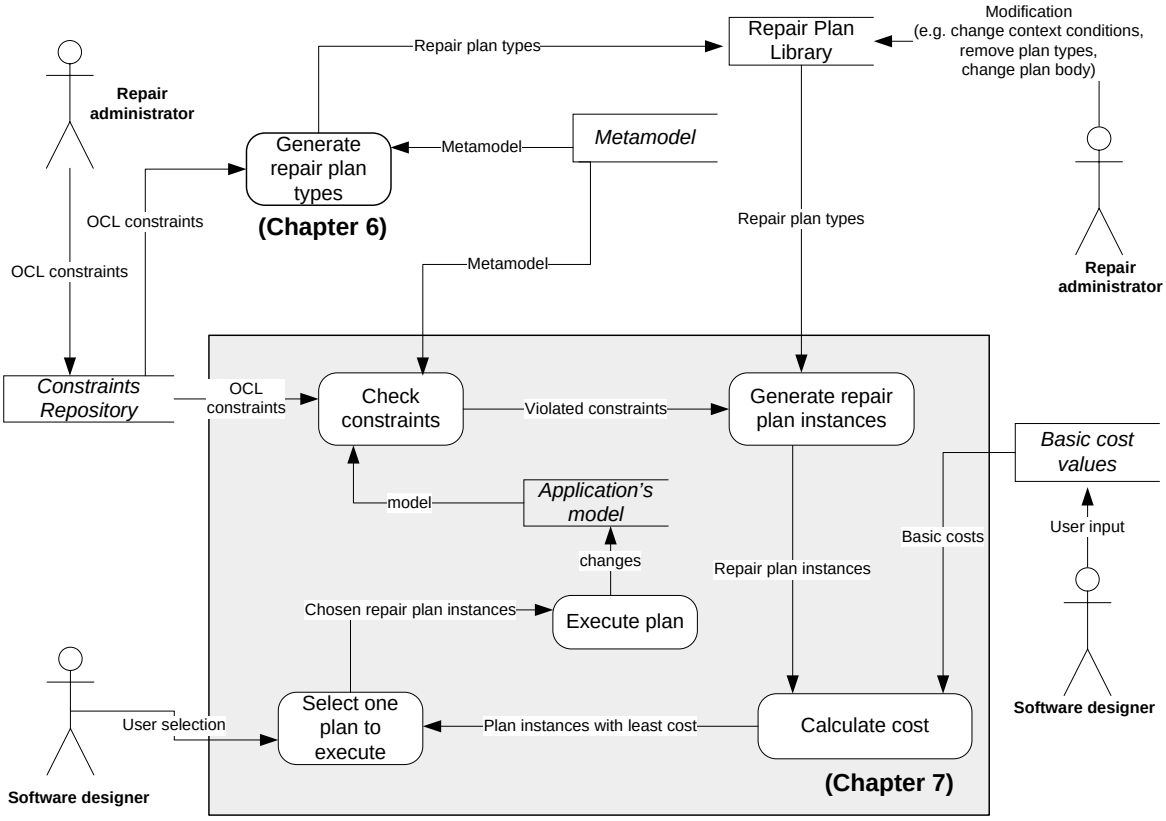


Figure 3.6: Change propagation framework

Secondly, the design model should be in a form that can be accepted as an input to a constraint checker that is used. Otherwise, a transformation is needed to translate the design model to a form that is accepted by the constraint checker. We will discuss this issue in more detail in chapter 8, where we present an implementation of our framework.

- A **metamodel** to which the application's design model should conform. As previously discussed in section 3.1.2, a metamodel is the central item on which consistency relationships can be defined. Depending on the types of design that we target, a metamodel can be available or not. For example, if one wants to apply our framework to UML models, they can use the existing UML metamodel [Object Management Group, 2005] or an excerpt of it. In contrast, the Prometheus methodology does not have a metamodel and consequently we needed to develop a metamodel for Prometheus, which is discussed in chapter 4.

- A set of **consistency constraints** that are expressed on the input metamodel. The *repair administrator*[6], creates consistency constraints and expresses them in OCL. For instance, UML has a set of well-formedness constraints that maintain the consistency of UML syntax and semantics [Object Management Group, 2005]. Those constraints can be input to our framework if one wishes to apply it to UML design models. For Prometheus, we have also developed a set of consistency constraints which is explained in detail in chapter 4.

Above are the three important inputs of the change propagation process. There are two important properties of change propagation: (a) it is cascading, i.e. performing an action to fix an inconsistency can cause further inconsistencies which require further actions; and (b) it has multiple choices, i.e. there are usually many ways of making the design consistent again. Those two properties are interestingly similar to the characteristics of the well-known and studied Belief-Desire-Intention architecture [Rao and Georgeff, 1992], which we have discussed in section 2.2.2 (page 30). In fact, BDI agents operate in an event-triggered manner, where events trigger plans, which in turn can create new events resulting in further plans being triggered. This is similar to the cascading nature of change propagation in which the need to fix an inconsistency matches an event and repair actions match a plan. Furthermore, in the BDI model an event can have multiple plans that it can trigger, with plan selection being made at run-time. This allows us to represent multiple ways of resolving a given inconsistency as separate plans, with the choice between them being specified in the plans' context conditions.

Based on that observation, in our framework a goal to repair a violated constraint (an inconsistency) is represented as an event and the ways to fix the violated constraint are represented as (repair) plans. For example, consider the constraint "constr(A, P)": *an agent A should play at least one role that handles a given percept P* (this constraint is a sub-constraint of constraint 1 we presented earlier in section 3.1.2). As can be seen in figure 3.7, there are several different plans that are triggered by the same "Repairing $constr(A, P)$" event. The first repair plan attempts to make an existing role played by agent A handle percept P. The second repair plan aims to make agent A play a role R (this role was not

[6]There are two actors interacting with our framework and they have different roles. The *repair administrator* is responsible for defining a metamodel and a set of constraints. Meanwhile, the *software designer* assumes the infrastructure (e.g. metamodels, constraints) is already available and just simply uses the tool to propagate changes in his working (design) model. In practice, the two roles can, however, be played by the same user.

played by the agent before) and to make R handle percept P. The third repair plan is similar to the second repair plan except that it creates a new role instead of using an existing one. It is important to note that those repair plans are *plan types*. At run time, each plan type can generate multiple plan instances depending on its *context condition*. For instance, the first repair plan in the above example can generate various repair plan instances, one for each of the existing roles played by agent A.

Plan #1: Use existing roles played by the agent
Triggering event: Repairing $constr(A, P)$
Context condition: R is a role played by agent A
Plan body:
1. Connect role R and percept P.

Plan #2: Use existing roles not played by the agent
Triggering event: Repairing $constr(A, P)$
Context condition: R is a role not played by agent A
Plan body:
1. Connect agent A and R.
2. Connect role R and percept P.

Plan #3: Create a new role
Triggering event: Repairing $constr(A, P)$
Context condition: None
Plan body:
1. Create role R.
2. Connect agent A and R.
3. Connect plan R and percept P.

Figure 3.7: Example of repair plans for fixing $constr(A, P)$

The repair plan's context condition is also used to determine whether a repair plan is applicable to handle a specific event. For example, if agent A does not play any roles, then the first repair plan is not applicable. Although an event can potentially trigger several applicable plans, only one of them will be executed to handle a particular event. Additional criteria can be incorporated in the context condition such as design heuristics, user preferences, etc. We can also have a default plan which involves the intervention of the user. This corresponds to the situation where none of the provided repair plans are applicable. This also ensures that there is always at least one applicable plan. However, we do not have this type of default

$$
\begin{array}{lcl}
P & ::= & E[:\, C] \leftarrow B \\
C & ::= & C \vee C \mid C \wedge C \mid \neg\, C \mid \forall\, x \bullet C \mid \exists\, x \bullet C \mid \mathit{Prop} \\
B & ::= & \mathit{RepairAction};\ \mid !E \ \mid\ B_1;\ B_2 \mid \\
 & & \mathit{if}\ C\ \mathit{then}\ B \mid \mathit{for\ each}\ x\ \mathit{in}\ SE\ B
\end{array}
$$

Figure 3.8: Repair plan abstract syntax

plan in our change propagation framework since we have a translation schema that generates a complete set of repair plans (i.e. always guarantees there is at least one applicable repair plan). This issue will be further discussed in chapter 6.

The syntax for repair plans (see figure 3.8[7]) is formally defined based on AgentSpeak(L) [Rao, 1996]. Each repair plan, P, is of the form $E : C \leftarrow B$ where E is the triggering event; C is an optional "context condition" (Boolean formula) that specifies when the plan should be applicable[8]; and B is the plan body, which can contain sequences ($B_1;\ B_2$) and events which will trigger further plans (written as $!E$). We extend AgentSpeak(L) by allowing the plan body to contain primitive repair actions that are presented in section 3.2.2, and also to contain conditionals and loops.

Below is how the repair plans in figure 3.7 are represented using the syntax defined in figure 3.8. We use $\mathit{constr}_t(A, P)$ to denote the event of making constraint $\mathit{constr}(A, P)$ true. We also define the abbreviation SE as the set of roles played by agent A.

P1 $\mathit{constr}_t(A, P) : R \in SE \leftarrow$ Connect role R and percept P

P2 $\mathit{constr}_t(A, P) : R \in \mathit{Set}(\mathit{Role}) \wedge R \notin SE$
$\leftarrow$ Connect agent A and R ; Connect role R and percept P

P3 $\mathit{constr}_t(A, P)$
$\leftarrow$ Create role R ; Connect agent A and R ; Connect role R and percept P

In this change propagation framework, the repair plans are generated automatically (at design time) from the constraints and metamodel, and form a *repair plan library* which is used at run time. One key consequence of generating plans from constraints, rather than writing

[7] "Prop" denotes a primitive condition such as checking whether $x > y$ or whether $x \in SE$.

[8] In fact when there are multiple solutions to the context condition, each solution generates a new plan instance. For example, if the context condition is $x \in \{1, 2\}$ then there will be two plan instances.

them manually, is that, by careful definition of the plan generation scheme, it is possible to guarantee that the plans generated are correct, complete, and minimal, i.e. there are no repair plans to fix a violation of a constraint other than those produced by the generator; and any of the repair plans produced by the generator can fix a violation. However, we also allow the repair administrators to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed. Since the library of plans is derived before runtime, the efficiency of deriving it is not crucial. In chapter 6 we discuss the (repair) plan generator in more detail.

The generation of repair plans is done ahead of time. At runtime, i.e. when the change propagation process starts (described as the shaded area in figure 3.6 on page 59), after primary changes are made to the design model, it is checked for inconsistencies. Our current implementation simply performs an exhaustive check of all the constraint instances that we take into account (i.e. the *repair scope*). However, better constraint checking strategies discussed in section 2.1.4 (on page 23) can be used in this process. For instance, we can use the information associated with the primary changes (e.g. the entity or relation being modified) to perform an incremental validation, which is more efficient because only the context of the last change, (instead of the whole model) is revalidated [Haesen and Snoeck, 2004; Wagner et al., 2003]. Other advanced constraint checking techniques such as the one proposed by Egyed [2006] can also be used to improve the performance.

Constraint checking may result in the detection of a number of violated constraints. When such a violation occurs, in order to have the constraint repaired a corresponding event is generated. A given constraint repairing event may trigger a number[9] of possible repair plans instances (instantiated from the library of repair plans). The determination as to which repair plans to choose is generally a design decision, which can be dependent on various factors such as the cause of inconsistencies, or even factors other than consistency that contribute to a good design (e.g. experience, knowledge on the future evolution of the design, design styles, etc.). In general, many of these dependencies may not even be capable of being formulated formally and being captured without extra knowledge provided by the software designer. As a result, it is expected that the execution of repair actions requires user interaction.

However, in some cases the number of different ways of fixing a inconsistency can be

[9]Which will be always be greater than zero, due to the way in which plans are generated

very large. Therefore, it is also important not to overwhelm the user with a large number of choices. For example, it is necessary to prevent infeasible repair options (e.g. repair actions that result in infinite cycles) from being presented to the user. In addition, it would be advantageous if the user had a simple mechanism to adjust the change propagation process, for example, specifying preference to bias the change propagation process towards adding more information rather then removing things.

We address the issue of repair plan selection by defining a suitable notion of *repair plan cost* that takes into account the important cascading nature of change propagation and fixing inconsistencies. Our framework has a cost calculation component that is responsible for calculating the cost of each repair plan instance. We recognize that fixing one violated constraint may also repair or violate others as a side effect, and so the cost calculation algorithm computes the cost of a given repair plan instance as including the cost of its actions (using basic costs defined by the software designer), the cost of any other plans that it invokes directly, and also the cost of fixing any constraints that are made false by executing the repair plan. In order to do this we simulate the BDI event-triggered change propagation mechanism in a manner similar to the lookahead planning technique (e.g. [Blythe, 1999; Erol et al., 1994]). The cost calculation component collects cheapest repair plans with the assumption that they are preferable from the perspectives of the user. We allow the user (i.e. the software designer) to specify the repair cost for each basic repair action. The software designer may use this mechanism to adjust the change propagation process. For example, if he/she wishes to bias the change propagation process towards adding more information then he/she may assign lower costs to actions that create new entities or add entities, and higher costs to actions that delete entities. In chapter 7, we will discuss our cost-based approach for repair plan selection in more detail.

The least cost plans (which can be more than one) are presented to the designer for selection. In case there is a large number of equal least cost plans, a certain heuristic can be used to make it faster for the software designer, by placing plans that are more likely to be chosen earlier in the list. For instance, we have implemented[10] a heuristic which use primary changes as input. More specifically, we identify new model entities that have been created by primary changes. We then sort the returned (cheapest) repair plans on the basis of the number of their contained repair actions involving those new entities. Repair plans

[10]This heuristic has not, however, been thoroughly developed or evaluated.

that have more actions affecting the new entities appear earlier in the list presented to the software designer. This heuristic assumes that "desirable" secondary changes likely target at the newly created entities so that they form consistency relationships with other existing entities in the model. This is because newly created entities usually cause inconsistencies due to the lack of relationships with other existing entities. In our view, it is reasonable for a heuristic to make a certain assumptions as long as they seem to work. Finally, it is noted that the user can choose not to make any selection, in this case he/she continues performing further primary changes and invokes the change propagation process later.

3.4 Chapter summary

This chapter serves to lay out a foundation for our work, which is the proposal of a novel, agent-based, change propagation framework. This framework tackles the important research questions that were outlined earlier in chapter 1. More specifically, we have explained how consistency relationships can be identified and represented using a metamodel and a set of consistency constraints. In this context, the change propagation process is based on fixing inconsistencies. We have also argued that the concepts of BDI agents are naturally suitable to represent and implement the change propagation mechanism. More specifically, the use of BDI-style, event-triggered, plans matches well with the cascading nature of change propagation and with the property of having multiple ways of fixing an inconsistency. We also briefly explain how we deal with the issue of automatically generating repair plans, and how to select between different applicable repair plan instances to fix a given constraint violation. These issues will be further discussed in the latter parts of this dissertation.

Our work is loosely related to the constraint satisfaction problems (CSPs) [Hentenryck and Saraswat, 1996]. CSPs address the combinatorial problem in which given a set of constraints among variables and a set of (domain) values the variables can take, what choices best satisfy these constraints. There have been some efforts on translating a UML model in terms of a class diagram annotated with OCL constraints into a CSP (e.g. [Cabot et al., 2008; Cadoli et al., 2004]) in order to automatically verify if the model is satisfiable, i.e. it is possible to create a correct (no model constraint violation) and non-empty instantiation of the model. However, the formulating process of mapping UML concepts (e.g. classes, associations, attributes, etc.) to CSP's variables and their domain values is complicated and challenging, especially maintaining the traceability between the original UML model and its

corresponding CSP. In addition, while a CSP constraint typically identifies the variables explicitly, an OCL constraint does not identify variables directly but instead gives navigation instructions for the UML model [Egyed, 2006]. Furthermore, constraint satisfaction techniques are often used on a finite domain for variable values whereas fixing inconsistencies in design models generally deals with open range values in which new values may be created and/or user intervention is often needed. Nonetheless, UML model checkers built in part on CSP technology such as UML2CSP [Cabot et al., 2007] can be used to pre-evaluate the inputs to our framework. More specifically, they can be used to check several correctness properties about the UML metamodel annotated with OCL constraints such as the satisfiability of the metamodel or the existence of contradictory constraints.

In the next two chapters, we will demonstrate how our framework is generic and applicable to a wide range of software engineering methodologies and design types. In particular, we will discuss how our framework can be applied to Prometheus, a prominent agent-oriented methodology (chapter 4), and to UML, a widely used modelling language for developing object-oriented systems (chapter 5).

Chapter 4

Case Study I: Prometheus

In the previous chapter, we have presented an agent-based change propagation framework that supports software designers in maintaining their designs. We have also argued that this framework is applicable to a range of software engineering methodologies and design types. In order to show this, we have performed two case studies involving the application of our framework to the Prometheus methodology, a prominent agent-oriented methodology, and UML, a well-known standard for modelling object-oriented designs. In this chapter, we discuss the first case study by beginning with a brief overview of the Prometheus methodology (section 4.1). In doing so, we explain the three major phases in Prometheus: system specification, architectural design and detailed design and describe the artefacts produced in each phase. Although Prometheus provides detailed processes, techniques, and heuristics for developing an agent-oriented system, the modelling language of Prometheus is not formally defined using a metamodel. On the other hand, a key data item required by our framework is a metamodel. Therefore, a step towards applying our approach to Prometheus is developing a metamodel for it. In section 4.2, we discuss a multi-layer metamodelling framework where a metamodel of Prometheus can be situated. Section 4.3 serves to provide details of the Prometheus metamodel that we have developed. In addition to a metamodel, our framework also requires a set of consistency constraints. The details of a set of consistency constraints for Prometheus, many of which have been developed independently of this work, are presented in section 4.4.

4.1 Overview of the Prometheus methodology

Prometheus[1] is a prominent agent-oriented software engineering methodology which has been used and developed over a number of years. The methodology is complete, described in considerable detail, and has tool support, i.e. Prometheus Design Tool (PDT[2]). The description in this section is necessarily extremely brief, and for further details we refer the reader to [Padgham and Perepletchikov, 2007; Padgham and Winikoff, 2004].

The Prometheus methodology consists of three phases: *system specification*, *architectural design* and *detailed design*. In this section, we describe them briefly with the main focus being on the artefacts that are produced at each stage. An overview of the methodology is depicted in figure 4.1.

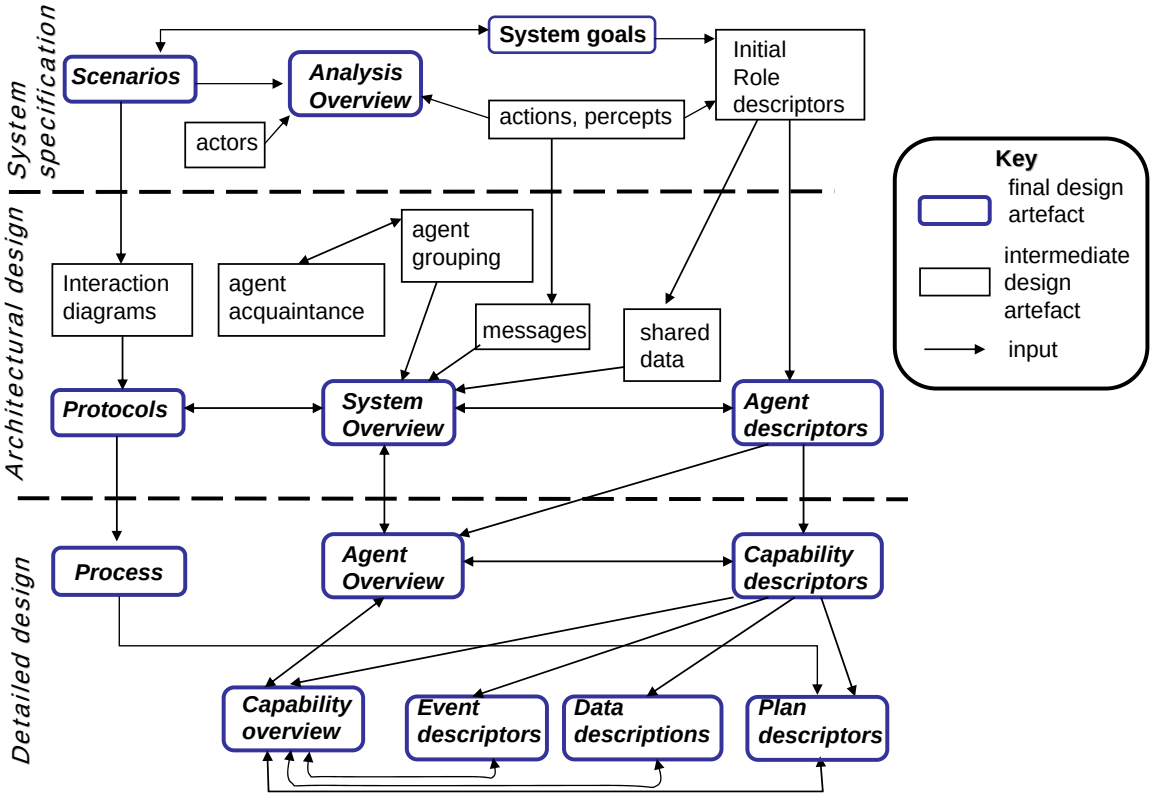


Figure 4.1: The Prometheus Methodology (obtained from the authors of Prometheus.)

[1]Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire (www.greekmythology.com).

[2]http://www.cs.rmit.edu.au/agents/pdt

4.1.1 System specification

The system specification is the first phase of Prometheus which involves the following activities:

1. Building a system's environment model.
2. Identifying *system goals* and sub-goals.
3. Developing use case *scenarios* illustrating the system's operation.
4. Grouping goals into the basic *roles* of the system

We now explain each of these activities in detail. As we described in chapter 2, one of the key properties of agents is "situatedness", which means that agent systems are situated in a dynamic environment. Therefore, developing the environment model is an important step in this system specification phase. Modelling an environment involves a very important activity: identifying *actors* and their interaction with the system. The concept of actors in Prometheus is similar to that of object-oriented analysis. Actors are any stakeholders who will interact with the system to achieve some goals, and can be humans or other software systems. For each actor, *percepts* which are inputs from the actor to the agent system are identified. In addition, outputs from the system to actors (*actions*) are identified. An *analysis overview diagram* is used in Prometheus to describe the relationships between actors, percepts, actions and scenarios.

In describing Prometheus we use an example system that is a simplified version of an actual application developed using Prometheus and reported in [Mathieson et al., 2004]. The simplified version we use is taken from the work of Jayatilleke [2007] and from teaching materials developed by the RMIT Agent Group[3], with some modifications by ourselves. Figure 4.2 shows an example of an analysis overview diagram for the simplified version of a weather alerting system[4] described in [Mathieson et al., 2004]. The "Forecaster" actor interacts with the system by providing "TAF[5]" percepts that contain forecasting data. The "Sensor (at AWS[6])" actor provides AWS inputs (i.e. actual weather data) to the system via an

[3] http://www.cs.rmit.edu.au/agents

[4] More description and discussion of this application can be found in section 9.4 of chapter 9. The analysis overview diagram is taken from a student project in the agent-oriented programming course taught at RMIT University.

[5] TAFs stand for terminal aerodrome forecasts, highly abbreviated forecasts of weather around airports intended for pilots [Mathieson et al., 2004].

[6] AWSs stand for automated weather stations.

"AWS" percept. It also receives requests from the system in the form of a "NewAWSRequest" action. The "Airport Official" interacts with the system in several ways: providing details of subscription ("ChangeSubscription" percept) for receiving weather alerts, and receiving response messages from the system ("PrintMsg" and "ConfirmSubscriptionUpdate" actions). The analysis overview diagram also shows which percepts are required by which scenarios, e.g. the "Provide Alert" scenario includes the "TAF" and "AWS" percepts, and the "Subscription Registration" scenario includes the "ChangeSubscription" percept. The purpose of these scenarios are discussed ahead.

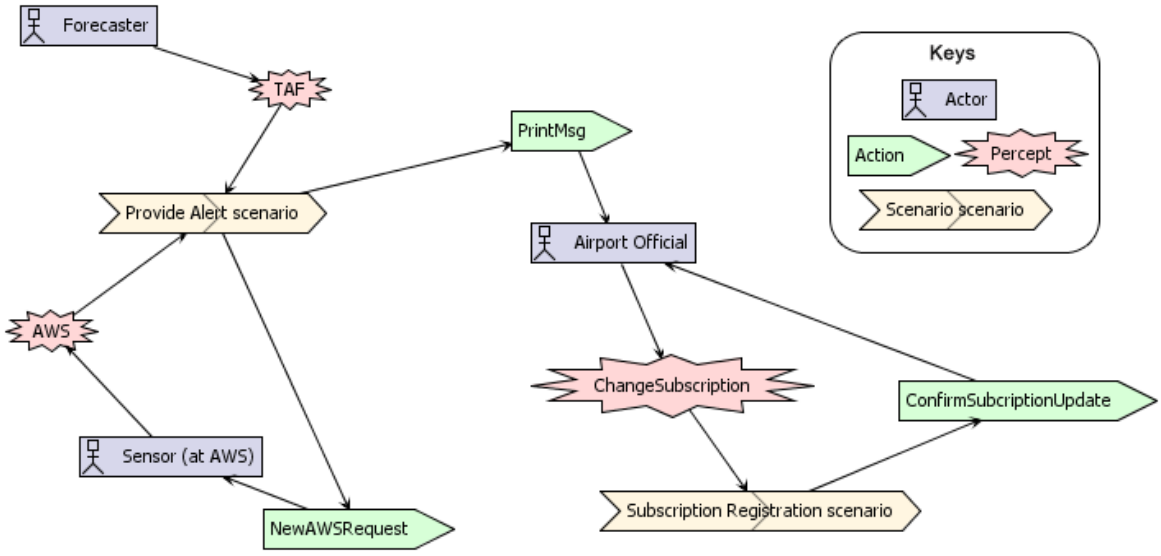


Figure 4.2: An analysis overview diagram for a weather alerting system

Similarly to identifying use cases in the object-oriented approach, the interaction between each actor and the system is described using scenarios in Prometheus. Each interaction scenario is described in a structured form which includes a sequence of steps, where each step can be an action being performed by a role, a percept being received by a role, a goal being achieved by a role, or a sub-scenario[7].

The system goals are identified on the basis of the initial scenarios as described above. Further goals are then elicited using abstraction and refinement techniques, as well as by developing scenario steps. This results in a goal hierarchy which is represented in a *goal overview diagram*. Figure 4.3 shows an example goal overview diagram for a weather alerting

[7]There is also an "other" step type which can be used to represent miscellaneous things such as waiting for a response.

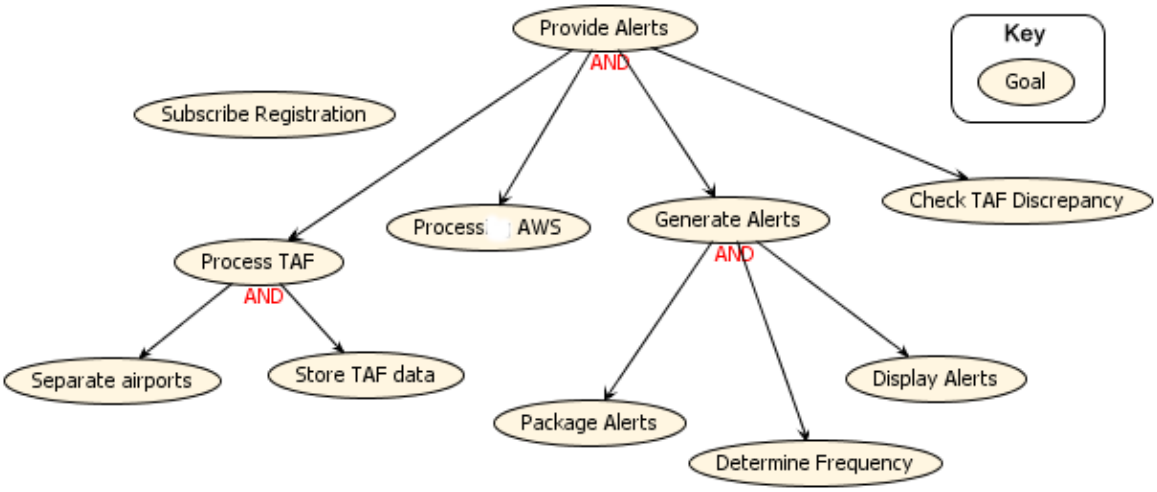


Figure 4.3: A goal overview diagram for a weather alerting system

system. As can be seen, the major goal of the system is providing alerts ("Provide Alerts" goal) which is broken down into several subgoals: "Process TAF", "Process AWS", "Check TAF Discrepancy", and "Generate Alert". These subgoals are also refined into sub-subgoals and so on.

The final step of the system specification phase involves identifying *roles*. Roles are obtained by grouping similar goals, and also including the percepts and actions associated with the included goals. A *role diagram* is used to capture the roles, and their percepts, actions and goals. According to figure 4.4, there are six different roles in the system: "Manage Subscription" role is for storing active subscriptions and delivering alerts to the relevant subscribers; "User Interaction" role is for receiving change subscription requests from the user, displaying messages such as alerts, and confirming subscription update; "Manage TAF Data" and "Manage AWS Data" roles are for managing each of these data types available in the system; "Check Discrepancies" role is responsible for detecting discrepancies between AWS and TAF data; and "Filter Alerts" role is for combining alerts for a given GUI which occur almost at the same time, determining whether to deliver an alert and controlling the frequency of alerts sent to the user.

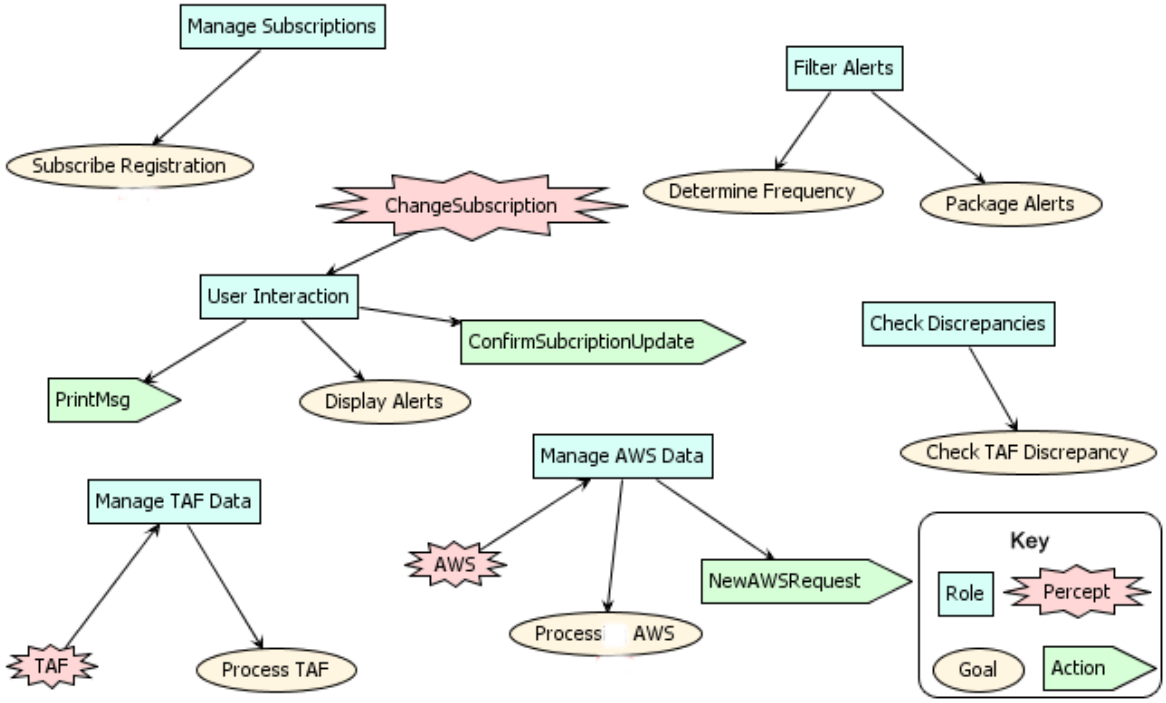


Figure 4.4: A role diagram for a weather alerting system

4.1.2 Architectural design

Between the system specification phase and the detailed design phase where the system is modelled as computational entities which are suitable for a particular agent platform, Prometheus provides an intermediate phase called architectural design. The major purpose of the architectural design phase in Prometheus is to identify the agent types within the agent system and the interactions between these agent types. The main steps of this phase are:

1. Determining what agent types will be implemented
2. Developing the interaction diagrams and interaction protocols that describe the dynamic behaviour of the system.
3. Developing the system overview diagram which captures the system's overall (static) structure.

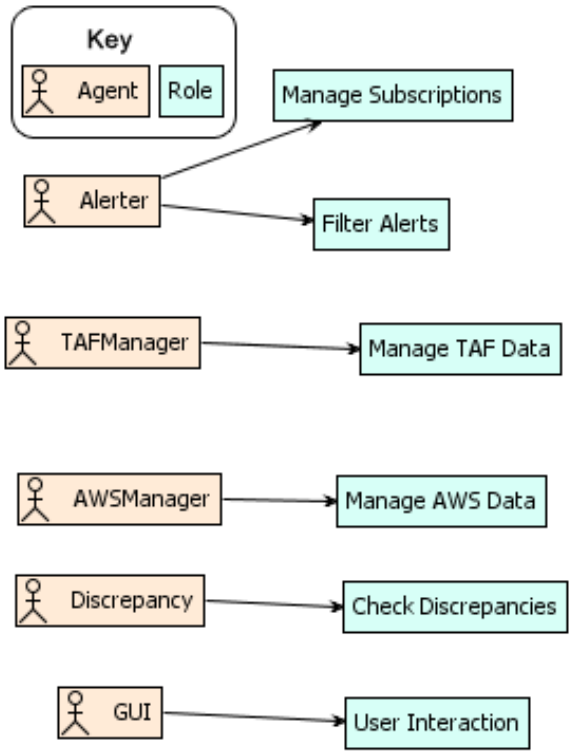


Figure 4.5: An agent role grouping diagram for a weather alerting system

Agent types are derived as groups of one or more roles. The choice of grouping is guided by considerations of coupling and cohesion. For instance, if there are significant interactions between two roles (e.g. a large number of messages exchanged), then there is a high possibility that they should be grouped (coupling criterion). On the other hand, roles that are related to each other (e.g. sharing the same data) are likely to be in the same group (cohesive criterion). Prometheus provides the *agent acquaintance diagram* and *data coupling diagram* to help the designer group roles into agent. The data coupling diagram shows the relationships between roles in terms of data used. Meanwhile, the agent acquaintance diagram shows interactions between agents as links. Figure 4.5 shows how roles are grouped into agents in the weather alerting system. In this case, each role is assigned to an agent except that the "Manage Subscription" and "Filter Alerts" roles are played by the "Alerter" agent.

Once the agent types have been determined, it is possible to start defining the interactions between them using *interaction protocols*. These protocols capture the dynamic behaviour of the system by defining the intended valid sequences of messages between agents. The

interaction protocols are developed from *interaction diagrams* which in turn are based on the scenarios developed in the system specification phase. Interaction protocols can be captured using a range of possible notations. The Prometheus methodology does not prescribe a particular notation, but the Agent UML (AUML[8]) notation is often used and is supported by the Prometheus Design Tool.

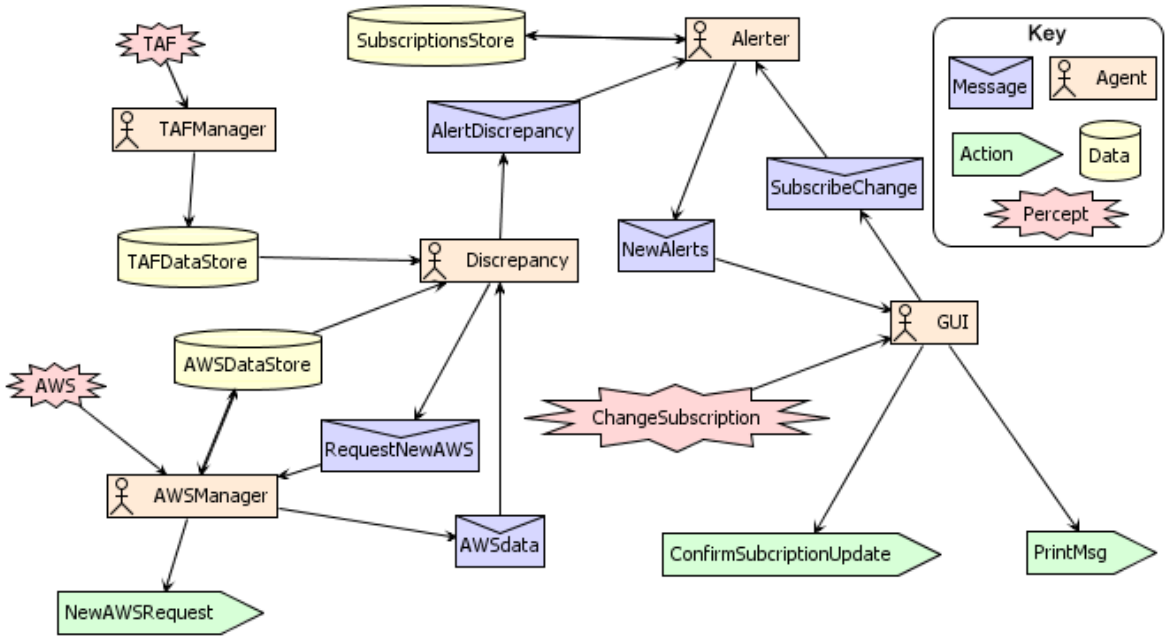


Figure 4.6: System overview diagram for a weather alerting system

The system's (static) structure is captured in a *system overview diagram*, which is regarded as the single most important design artefact in Prometheus. The system overview diagram gives the software designer a general picture of how the system as a whole is structured. This type of diagram shows the agent types, the communication links between them, and data. It also shows the system's boundary and its environment in terms of actions, percepts, and external data. For example, figure 4.6 shows the system overview diagram for the weather alerting system. It shows how the five agents in the system interact with each other. "TAFManager" and "AWSManager" agents process "TAF" and "AWS" percepts respectively, and store the relevant data which can be accessed by the "Discrepancy" agent. This agent sends "AlertDiscrepancy" messages to the "Alerter" agent, which uses the "SubscriptionStore"

[8]http://www.auml.org

data to work out which GUI agents should receive discrepancy alerts. A "GUI" agent can also register for being alerted by sending a "SubscribeChange" message to the "Alerter" agent.

4.1.3 Detailed design

The final stage of the Prometheus methodology is the detailed design. The internal structure of each agent and how it will accomplish its goals within the overall system are addressed in this phase. Specifying agent internals in Prometheus is a process of progressive refinement, including the following activities:

1. Defining and developing capabilities (modules within agents) and their relationships.
2. Developing process diagrams depicting the internal processing of each agent related to the protocol specifications. However, process diagrams are not as well developed in Prometheus, are less commonly used and are not supported by the Prometheus Design Tool. As a result, we do not cover process diagrams in this thesis.
3. Developing plans, events, and data and their relationship.

Most of the Prometheus methodology does not assume any particular agent architecture. However, the lowest layer of plans and events does assume that the target agent architecture is plan-based (which is the case for BDI agent platforms). This could easily be exchanged for an alternative architecture if desired, but a detailed design which is close to code, must make some assumptions about the implementation architecture. *Agent overview diagrams* and *capability overview diagrams* capture the structure of the capabilities, sub-capabilities, plans, events and data within the agent. Figure 4.7 shows the internals of the "Discrepancy" agent, which contain several plans to achieve its goal of detecting discrepancies. The 'HandleAwsDataPlan" plan is triggered by an "AWSData" message, sent by the "AWSManager" to inform it that a new AWS has just arrived. This plan then retrieves TAF and AWS data from "TAFDataStore" and "AWSDataStore" respectively and processes them. It may also request for new AWS by sending a "RequestNewAWS" message to the "AWSManager" agent. Discrepancy detection for each data type and sending alerts to the "Alerter" agent are done by "CheckTempDiscrepancy" and "CheckPressDiscrepancy" plans.

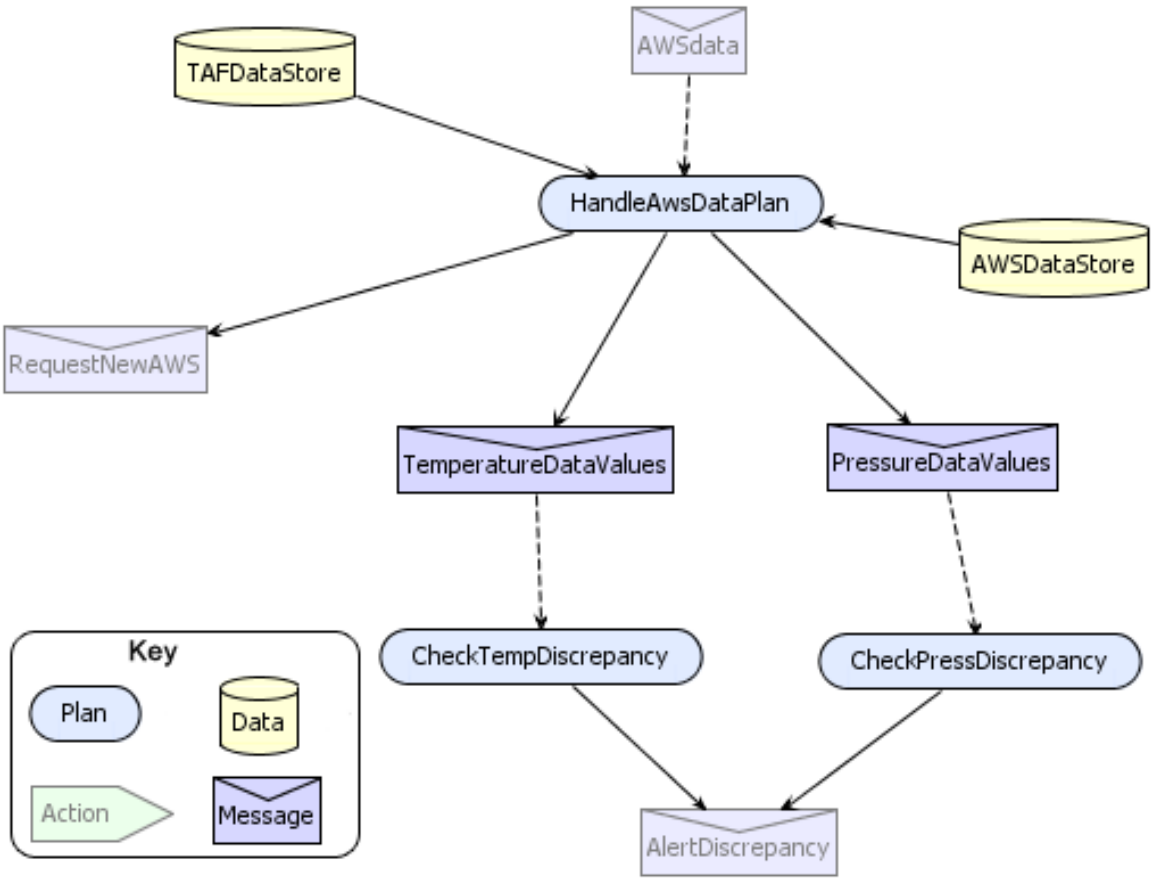


Figure 4.7: Agent overview diagram for the "Discrepancy" agent

4.1.4 Prometheus diagrams

As previously described, following the Prometheus methodology produces a range of artefacts at different stages of the software development lifecycle. These artefacts form a semantically consistent abstraction of an agent system to be built. Each artefact represents a different aspect or abstraction level of the underlying system and can be seen as a "view" on the full underlying model. Each view depicts various associations between the Prometheus concepts or entities, such as actors, goals, agents, roles, percepts, actions, etc. Many of the entities may appear in different views. For instance, the same percept entity can appear in the analysis overview diagram, a scenario editor, the role diagram, the system overview diagram, an agent overview diagram and a capability overview diagram. In each of these views, the percept has associations with other entities. This repetition of entities across different views

induces dependencies among them. Figure 4.8 summarizes the most important diagrams that appear in Prometheus or PDT.

Diagram	Purpose
System Specification	
Analysis Overview	How the system interacts with its environment in terms of actors, percepts, actions, and scenarios
Scenario Editor	Possible sequential interaction between roles
Goal Overview	Structure of system goals and subgoals
System Roles	Grouping of goals, percepts and actions into roles
Architectural Design	
Agent-Role Grouping	Grouping of roles into agents, derived from Data Coupling and Agent Acquaintance
System Overview	Overall system structure in terms of agent types, interactions between them and the environment
Protocol	Possible interaction between agents
Detailed Design	
Agent Overview	Internals of agents in terms of their capabilities, events, plans and data structures
Capability Overview	Structure of the plans and events within capabilities
Process	Internal processing of an agent

Figure 4.8: Prometheus diagrams

In this section, we have provided an overview of the Prometheus methodology. We have also summarized the different diagrams within Prometheus that provide structural and behavioural views of the system under development. In the next section, we review the multi-layer metamodel hierarchy approach and discuss where a metamodel of Prometheus will reside.

4.2 The four-layer metamodel hierarchy

Model driven development (MDD) generally advocates the multi-layer modelling approach. For example, the Model Driven Architecture (MDA) [Kleppe et al., 2003], the MDD framework part of OMG standards, uses the well-known four layer metamodelling framework for defining where entities are used within and between models. Therefore, an important step to develop a metamodel for Prometheus is defining where its metamodel is situated in a multi-layer metamodelling framework.

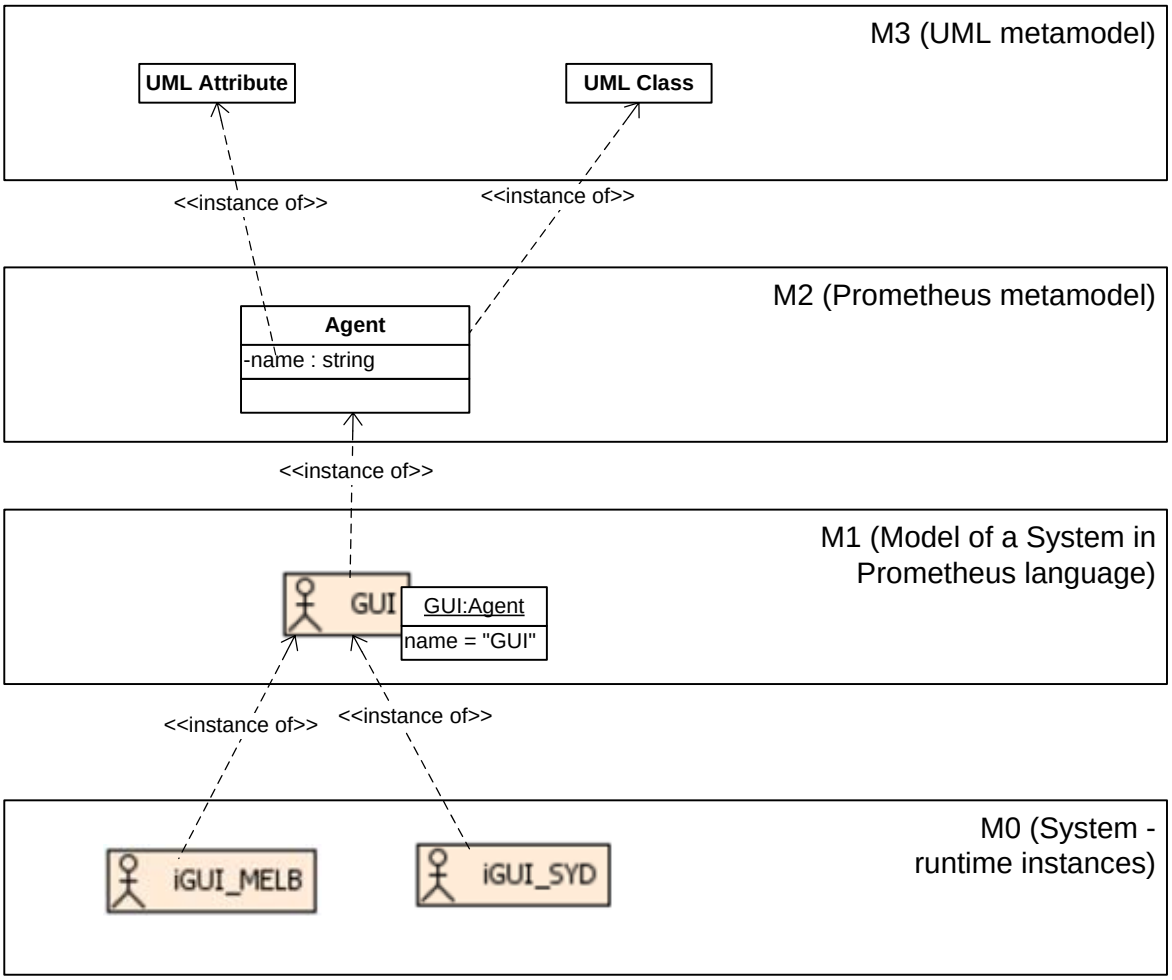


Figure 4.9: The four-layer metamodel hierarchy

Similarly to the MDA four-layer metamodel hierarchy, the lowest layer is called M0 (see figure 4.9). At this level, there is the running system that contains the actual ("real") instances. These instances are, for example, the agents named "iGUI_MELB" and "iGUI_SYD". Such instances are the run-time instances of model elements defining a specific domain, e.g. a weather alerting business.

In MDA context, the M1 layer contains models of a system. It is mainly responsible for defining languages that describes certain domains. Prometheus provides such a language to describe a domain from an agent-oriented perspective. A Prometheus model is an M1 model that contains, for instance, a "GUI" agent type, an "AlertDisrepancy" message type, a "CheckTempDiscrepancy" plan type. It is noted that each entity at the M0 layer is always

an instance of an element at M1 layer. For example, the agents named 'iGUI_MELB" and "iGUI_SYD" are instances of the M1 element, Prometheus "GUI" agent type.

The next higher level, also referred to as M2, contains elements that specify the elements at the M1 layer. More specifically, every element at M1 is an instance of an M2 element, and the concepts at M2 are all categorisation or classification of instances at M1. A metamodel for Prometheus that we want to build should reside in this layer. This metamodel should be defined in such a way that every Prometheus model (at M1) is an instance of the Prometheus metamodel (at M2). For instance, the "GUI" agent type in a Prometheus model is shown in figure 4.9 as an instance of the class "Agent".

As we use UML to define a metamodel for Prometheus, our next higher level, i.e. M3, clearly contains the UML metamodel. The same relationship that exist between elements of lower layers is present between elements of the M2 and M3 layers. Each element in M2 should be an instance of an element in M3. For instance, the "Agent" class (in Prometheus metamodel) is an instance of a "UML Class", and the "name" property of the "Agent" class is an instance of a "UML Attribute". It is noted that similar to MDA, we can have another higher layer that contains the Meta-Object Facility (MOF) to define the UML metamodel. In addition, we choose the UML metamodel (instead of MOF) as the metametamodel for the M2 Prometheus because the Dresden OCL2 Toolkit that we used for our implementation currently supports only UML models (see chapter 8 for more details). However, our main focus is at the M2 layer, where the Prometheus metamodel resides.

4.3 A metamodel for Prometheus

A Prometheus metamodel should specify what can be expressed in the valid models of the Prometheus modelling language. It should also formally capture the relationships between all Prometheus concepts such as actors, goals, agents, roles, percepts, actions, etc. In [Jayatilleke, 2007], a metamodel has been developed for Prometheus in order to support automatic transformations from Prometheus models to source code. However, the focus of that metamodel is on detailed design and implementation, and it does not cover some of the entities at system specification and architectural levels such as actors, scenarios, roles, etc. Consequently, we have developed a metamodel of Prometheus by extracting such well-formedness conditions and relationships between Prometheus entities presented in [Padgham and Perepletchikov, 2007; Padgham and Winikoff, 2004; Perepletchikov and Padgham, 2005]. Our

metamodel covers entities described across the three development phases in Prometheus, i.e. system specification, architecture design and detailed design.

As our Prometheus metamodel is relatively large, we present here its different snapshots. Figures 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16 provide different snapshots of the main metamodel showing the relationships between those Prometheus entities.

4.3.1 ModelEntity

A ModelEntity represents all the Prometheus entities, i.e. all these entities are directly or indirectly inherited from the ModelEntity. Each Prometheus model entity has a name, a description, a unique identification, and some notes attached to it.

Attributes

name	The name of the entity.
description	A brief description of the entity.
uniqueId	A unique identifier for this entity
notes	Some notes related to this entity.

4.3.2 Actor

An actor represents an external entity, which can be human or software systems (see figures 4.10 and 4.15). In the metamodel, an Actor may affect the system through a set of Percepts. The system in turn may affect an Actor through Actions. An Actor may also participate in a number of Scenarios and Protocols.

Associations

action	Actions produced by a system which the Actor interacts with. These actions give certain effects on the Actors.
percept	The set of percepts represented how the Actor input to the system that it interacts with.
scenario	The set of use case scenarios in which this Actor participates.
protocol	The set of protocols in which this Actor participates.

4.3.3 Role

A role represents a certain functionality of a system. Roles are developed by grouping goals based on their functional relatedness. In the metamodel (see figures 4.10, 4.11, and 4.13),

a Role can handle several Percepts, perform a number of Actions, and is responsible for achieving Goals. A Role may also have access (read or write) to some Data. Steps in a Scenario have to be associated with at least one Role. Finally, a Role should be played by at least one Agent.

Associations

perceptsEntityReference	The set of percepts that are handled by the Role.
actionsEntityReference	The set of actions that are performed by the Role.
goalsEntityReference	The set of goals that are achieved by the Role.
readData	Data that are read by the Role.
writtenData	Data that are written by the Role.
step	The steps (in a particular scenario) that are performed by the Role.
agent	One or more agents that play the Role.

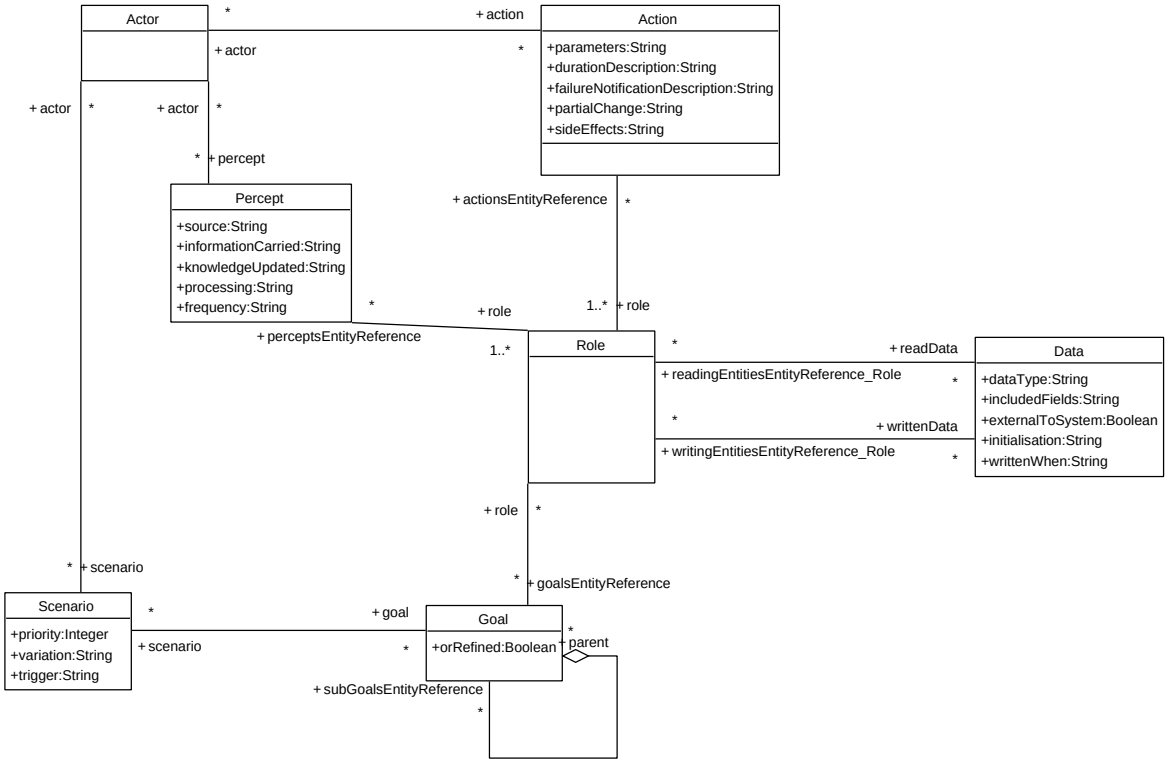


Figure 4.10: Metamodel snapshot relating to system specification entities

4.3.4 Percept

Percepts represent inputs from an actor or environment to the system. In the metamodel (refer to figures 4.10, 4.11, 4.12, 4.14 and 4.16), a Percept can be provided by several Actors. A Percept can be linked to a PerceptStep in a scenario. A Percept should be handled by at least one Role and one Agent. Capabilities are also able to handle Percepts. The system processes its input Percepts using Plans. In some cases, Percepts are also the events that trigger some Plans.

Attributes

description	Indicates the situation in which this Percept is received.
source	Describes how information is retrieved from the environment.
informationCarried	Specifies what information is available regarding the Percept.
knowledgeUpdated	Identifies knowledge updates resulting from the Percept. This can be either directly or indirectly (via some forms of reasoning) extracted from the information carried.
processing	Indicates how the Percept is processed or is received directly from the agent sensor.
frequency	Specifies the likely frequency of the Percept. It may contain extra information relating to handling of extreme cases if necessary.

Associations

actor	Specifies the actors who provides the given Percept.
role	The set of roles that are responsible for handling the given Percept. At least one role are assigned to handle the Percept.
step	Indicates the (scenario) steps that this Percept is associated with.
agentRespondingEntityReference	One or more agents that respond to this Percept.
capability	Capabilities that respond to this Percept.
plan	The set of plans that process this Percept.
triggeredPlan	Plans that are triggered by this Percept. In this case, the Percept is considered as an event.

4.3.5 Action

Actions represent outputs from the system to actors. In the metamodel (see figures 4.10, 4.11, 4.12, 4.14 and 4.16), an Action can give effects on a number of Actors. An Action is linked with an ActionStep in a scenario, and is performed by at least one Role and one

Agent. Capabilities are also able to perform Actions. An Action must be performed by at least one Plan.

Attributes

description	Describe the action and its intended results.
parameters	Specifies parameters that can influence how the action is performed.
durationDescription	Indicates whether the action is instantaneous or durational.
failureNotifiationDescription	Indicates how failure is checked and reported.
partialChange	Describes the change that results from performing the failed action.
sideEffects	Indicates the side effects of the action.

Associations

actor	Specifies the actors who receives the Action from the system.
role	The set of roles that are responsible for performing the given Action.
step	Indicates the (scenario) steps that this Action is associated with.
agent	One or more agents that perform this Action.
capability	Capabilities that perform this Action.
plan	The set of plans that perform this Action.

4.3.6 Data

Data represents the persistent data that are used in the system. In the metamodel (see figures 4.10, 4.11, 4.12, 4.14 and 4.16), Data can be read or written by Roles, Steps in a scenario, Agents, Capabilities and Plans. Data can also be owned by an Agent or a Capability.

Attributes

dataType	Indicates the type of this Data.
includedFields	Indicates the fields that are contained in this Data.
externalToSystem	Indicates whether this data is external to the system or not.
initialisation	Describes the initialisation process or values.
writtenWhen	Specifies what the data is used for and when it is updated.

Associations

readingEntitiesEntityReference_Role	The roles that read this Data.
writingEntitiesEntityReference_Role	The roles that write to this Data.
stepReader	Indicates the (scenario) steps that read this Data.
stepWriter	Indicates the (scenario) steps that write to this Data.
agentOwner	One or more agents that own this Data. Instances of the given Data are internal to these agents.
writtenByEntityReference_Agent	One or more agents that write to this Data. The given Data is external to these agents.
readByEntityReference_Agent	One or more agents that read this Data. The given Data is external to these agents.
ownerCapability	One or more capabilities that own this Data. Instances of the given Data are internal to these capabilities.
writtenByEntityReference_Capability	One or more capabilities that write to this Data. The given Data is external to these capabilities.
readByEntityReference_Capability	One or more capabilities that read this Data. The given Data is external to these Capability.
planWriter	The set of plans that write to this Data.
planReader	The set of plans that read this Data.

4.3.7 Goal

Goals represents the purposes of the system to be developed. In the metamodel (see figures 4.10, 4.11, 4.12, 4.14 and 4.16), a Goal is assigned to Roles, Agents, Capabilities and Plans. A Goal is also associated with a Scenario that illustrates a means to achieve it. Similar to a percept and an action, a Goal can be linked to a Step in a scenario. A Goal can contains other Goals, which are usually referred to as subgoals.

Attributes

orRefined	Indicates whether this goal is OR-refined or AND-refined. If a goal is AND-refined, it means that subgoals are steps (that must be done) in achieving the overall goal. If a goal is OR-refined, then subgoals are alternative ways of accomplishing the goal, i.e. achieving any of them is sufficient.

Associations

role	The set of roles that are responsible for achieving the given Goal.
scenario	The scenarios that mean to achieve the goal.
subGoalsEntityReference	The set of subgoals of this Goal. Primitive goals, i.e. goals that are at the leaves of the goal tree, do not have any subgoals.
step	The (scenario) steps that this Goal is associated with.
agent	The set of agents that achieve this Goal.
capability	The set of capabilities that achieve this Goal.
plan	The set of plans that aim to achieve this goal.

4.3.8 Scenario

A scenario describes the interaction between actors and the system. In the metamodel (see figure 4.11), each Scenario is associated with a goal, which the scenario is one way of accomplishing. Different scenarios can be linked with the same goal. A scenario contains a number of Steps. Each Step in a scenario is performed by a number of Roles (typically one role) and can have access to some Data.

Attributes

priority	The priority of this scenario. A high number indicates a high priority.
variation	Describes certain variations of this scenario.
trigger	Indicates the triggers of this scenario.

Associations

actor	The actors who participate in this scenario.
goal	The goals that this scenario aims to achieve.
stepsEntityReference	A sequence of steps in this scenario.
step	The (scenario) steps that this scenario is associated with.

4.3.9 Step

Steps are primitive entities that form a scenario. In the metamodel (figure 4.11), a Step can be of different kinds: GoalStep, ActionStep, PerceptStep, ScenarioStep or OtherStep. A GoalStep is linked with a Goal, an ActionStep is linked with an Action, and a PerceptStep is linked with a Percept. ScenarioStep is used to specify a sub-scenario, and is linked to a scenario. OtherStep is to cover other cases such as waiting for something to happen. A Step

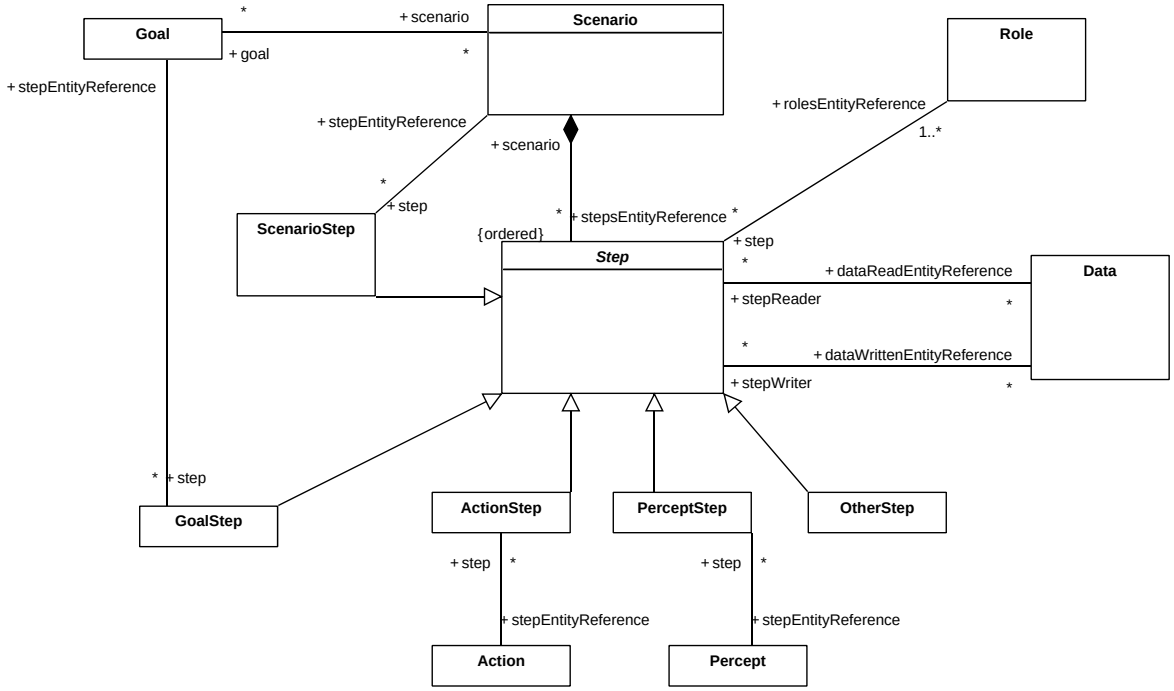


Figure 4.11: Metamodel snapshot relating to Scenario

is part of a Scenario and is assigned to a number of Roles (typically only one). A Step may also read or write some Data.

Associations

scenario	The scenario which owns this step.
rolesEntityReference	The roles which perform this step.
dataReadEntityReference	The data read by this step.
dataWrittenEntityReference	The data written by this step.

4.3.10 Agent

An agent is the most important entity in the system. In Prometheus, agents are formed by combining roles. In the metamodel (figures 4.12, 4.13, and 4.15), an Agent plays at least one Role. An Agent also handles some Percepts, performs some Actions, achieves some Goals, reads/writes to some Data, and sends/receives some Messages. An Agent may also own some internal Data or Messages (i.e. messages that are posted and handled internally to a single agent). An Agent may contain a number of Capabilities and Plans. An Agent may also take

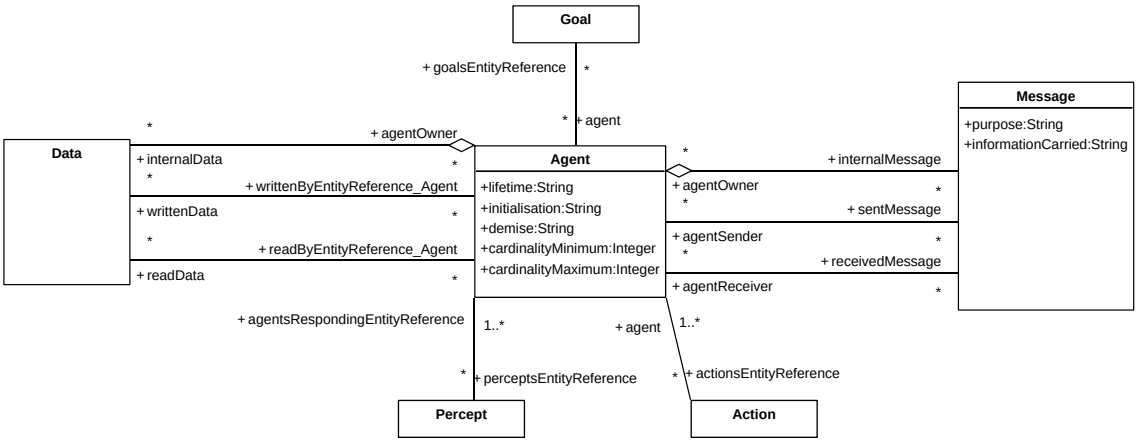


Figure 4.12: Metamodel snapshot relating to Agent

part in several protocols as a means of interacting with other agents.

Attributes

lifetime	Indicates the lifetime of the agent.
initilisation	Describes what is involved in the process of initiating this agent.
demise	Describes what is involved in the process of terminating this agent.
cardinalityMinimum	Indicates the allowable minimum number of instances of this agent.
cardinalityMaximum	Indicates the allowable maximum number of instances of this agent.

Associations

perceptsEntityReference	The percepts that this agent is responsible for handling.
goalsEntityReference	The goals that this agent aims to achieve.
actionsEntityReference	The set of actions performed by this agent.
internalData	Data owned by this agent.
writtenData	Data written by this agent.
readData	Data read by this agent.
internalMessage	Messages owned by this agent.
sentMessage	Messages sent by this agent.
receivedMessage	Messages received by this agent.
rolesEntityReference	One or more roles played by this agent.
includedPlansEntityReference	Plans owned by this agent.
includedCapabilitiesEntityReference	Capabilities owned by this agent.
protocolsEntityReference	Protocols that this agent takes part in.

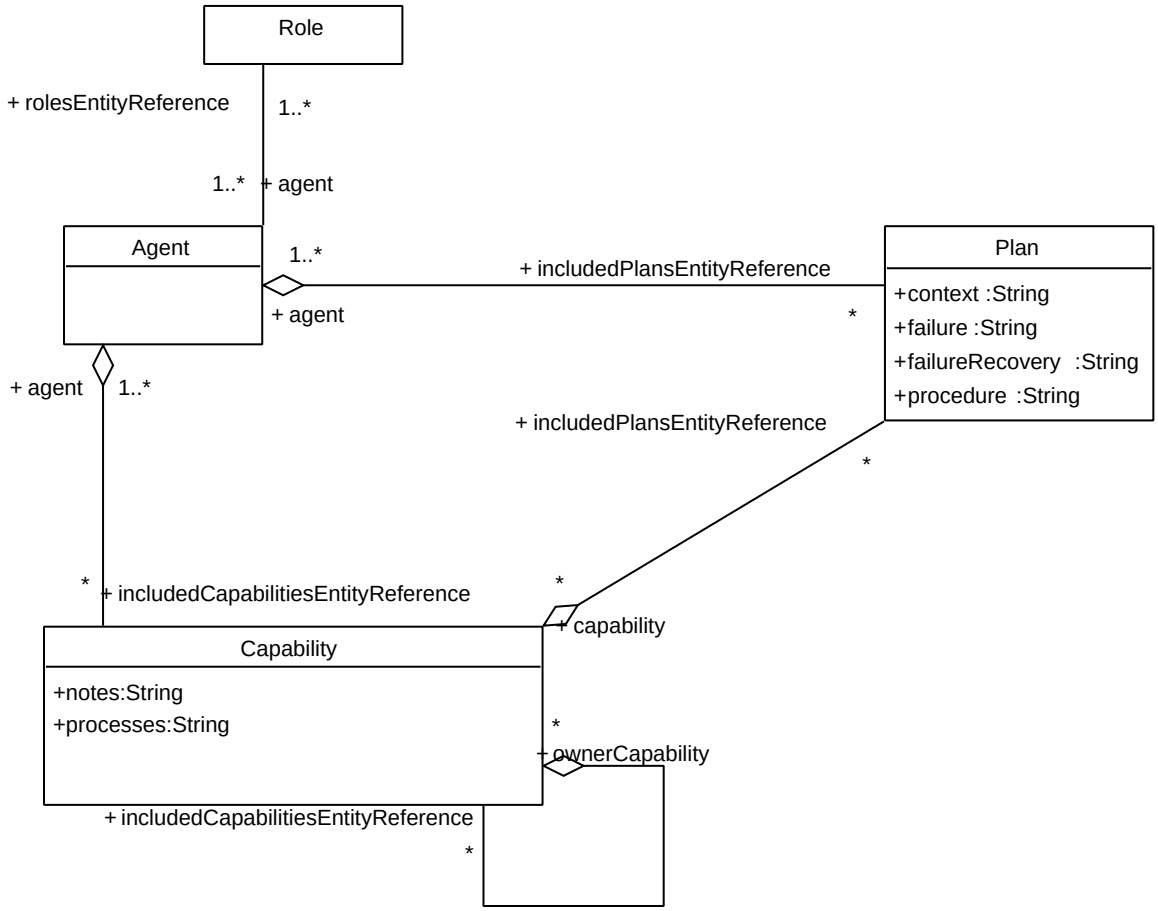


Figure 4.13: Metamodel snapshot relating to Role, Agent, Capability and Plan

4.3.11 Capability

Capabilities allow agent elements to be hierarchically structured. In the metamodel (figures 4.13 and 4.14), a Capability may handle some Percepts, perform some Actions, achieve some Goals and read/write to some Data. A Capability may also own some internal Data. A Capability may be nested within other Capabilities and may contain a number of Plans.

Attributes

process	Description of an internal process of this Capability.

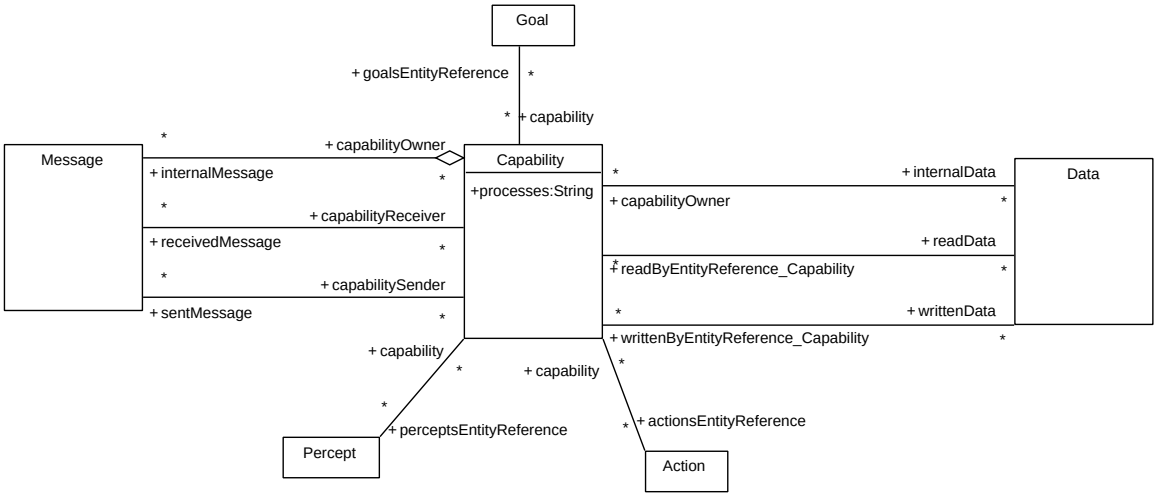


Figure 4.14: Metamodel snapshot relating to Capability

Associations

perceptsEntityReference	The percepts that this capability is responsible for handling.
goalsEntityReference	The goals that this capability aims to achieve.
actionsEntityReference	The set of actions performed by this capability.
internalData	Data owned by this capability.
writtenData	Data written by this capability.
readData	Data read by this capability.
internalMessage	Messages owned by this capability.
sentMessage	Messages sent by this capability.
receivedMessage	Messages received by this capability.
agent	The agents owning this capability.
includedPlansEntityReference	Plans owned by this capability.
includedCapabilitiesEntityReference	Capabilities owned by this capability.
ownerCapability	The parent capability of this capability.

4.3.12 Message

A message defines a particular communication between agent instances. In the metamodel (see figures 4.12, 4.14, 4.15 and 4.16), a Message is sent and received by Agents, Capabilities and Plans. Messages may be owned by an Agent or a Capability, i.e. posted (and handled) internally within the Agent or the Capability. A Message can also be an event which triggers

a Plan.

Attributes

purpose	Indicates the purpose of this message.
informationCarried	Information contained in this message.

Associations

agentOwner	The agents which own this message.
agentSender	The agents which send this message.
agentReceiver	The agents which receive this message.
capabilityOwner	The capabilities which own this message.
capabilitySender	The capabilities which send this message.
capabilityReceiver	The capabilities which receive this message.
planReceiver	The plans receiving this message.
triggeredPlan	The plans triggered by this message.
planSender	The plans sending this message.

4.3.13 Protocol

A protocol defines exactly which interaction sequences are valid within the system. Interaction protocols consists of sequencing, iteration, alternatives, and other control structures. There is a range of notations that can be used to describe protocols such as UML activity diagram, AUML[9] (Agent UML) [Odell et al., 2000], and Petri nets (e.g. [Nowostawski et al., 2001]; [Poutakidis et al., 2002]). A protocol in Prometheus is defined based on the revised version of Agent UML [Huget and Odell, 2004]. This version of AUML contains a range of notations for describing sequence diagrams, interaction overview diagrams, communication diagrams, and timing diagrams. AUML has been receiving a reasonable level of community acceptance. More specifically, its sequence diagrams notation has been adopted by several prominent agent-oriented methodologies (e.g. Prometheus, Gaia, and Tropos) for depicting agent interactions, and by FIPA[10] for describing standardised protocols.

The Prometheus Design Tool uses a textual notation for AUML to provide a precise, formal and simple definition of agent interaction protocols [Winikoff, 2007]. The metamodel that we have developed for protocol in Prometheus is based on that textual notation. In the metamodel (figure 4.15), a Protocol has participation from two or more Agents. A Protocol

[9] http://www.auml.org
[10] The Foundation for Intelligent Physical Agents, http://www.fipa.org.

can contain Pelements which can be either a Message, a Goto, a Label, a Box, a Region or even a SubProtocol. A Protocol may refer to SubProtocols.

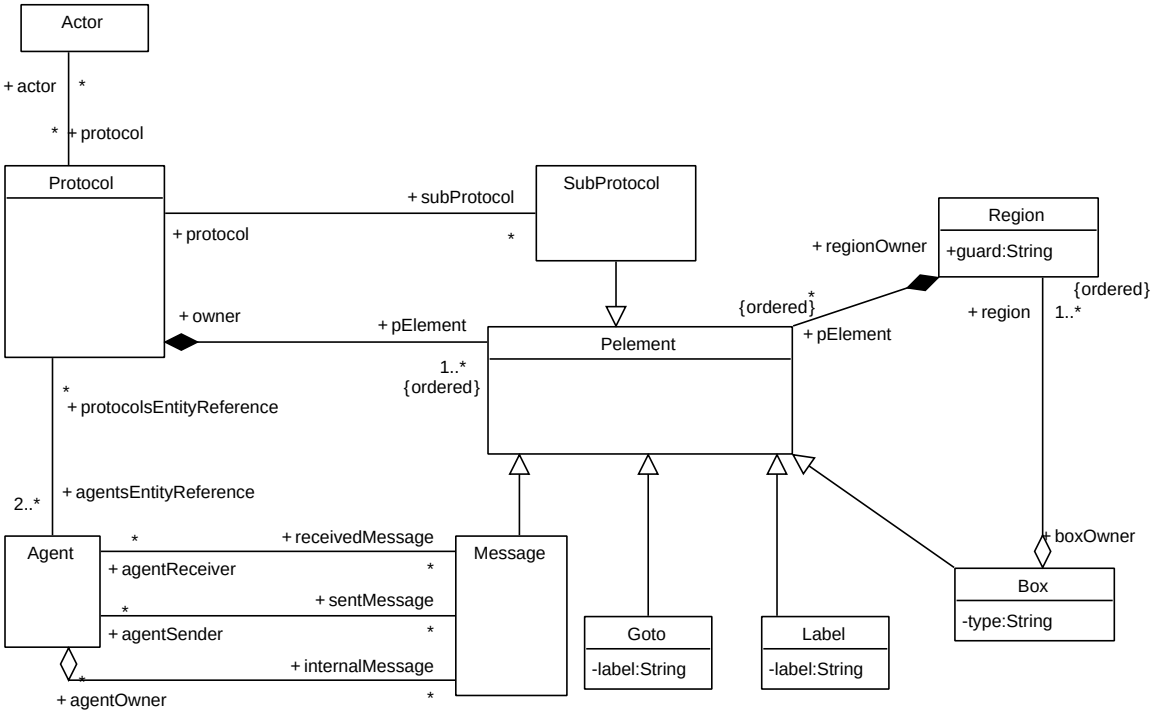


Figure 4.15: Metamodel snapshot relating to Protocol

Associations

subProtocol	The protocols that are included in this protocol.
pElement	The protocol elements.
agentsEntityReference	The agents which participate in this protocol.
actor	The actors who participate in this protocol.

4.3.14 Pelement

A protocol element describes what are the constituents within a protocol. In the metamodel, a Pelement is an abstract class that represents either a Message, a Goto, a Label, a Box, a Region, or a SubProtocol. A Pelement is contained in a Protocol and a Region. A Box can be divided into a number of Regions, which can contain Pelements. There are several types of Box such as "Alternative", "Option", or "Parallel", which are specified in the *type* attribute. Each region can contain Pelements and has a guard, specifying a condition on

that region being selected. Labels and Gotos represent incoming and outgoing continuations respectively. Finally, a SubProtocol represents a reference to another protocol. It is noted that every Pelement must belong to exactly one protocol or one region.

Associations

owner	The protocol which owns this element.
regionOwner	The region which owns this element.

4.3.15 Plan

A plan describes the details of how an agent or capability achieves its goal. In the metamodel (figures 4.13 and 4.16), a Plan is assigned to accomplish some Goals, respond to some Percepts, perform some Actions, read or write to some Data, and send and/or receive some Messages. A Plan is triggered by an event which is either a Percept or a Message. A Plan may be contained in some Capabilities and in at least one Agent.

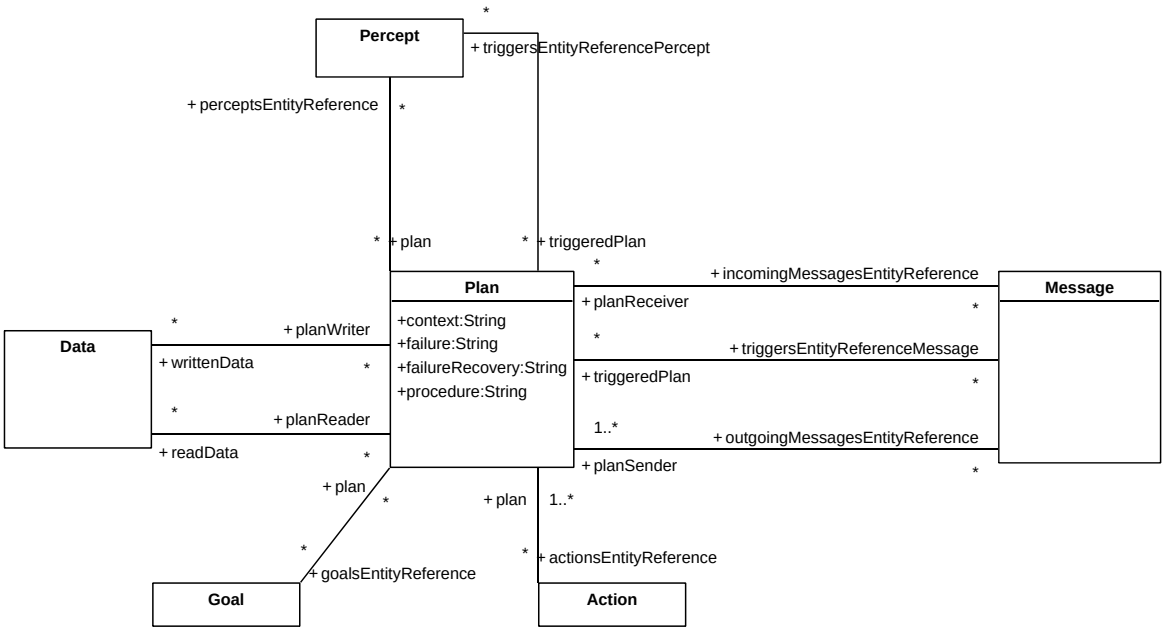


Figure 4.16: Metamodel snapshot relating to Plan

Attributes

context	Indicates the context condition of this plan.
failure	Describes situations where this plan fails.
failureRecovery	Describes how the plan recovers from a failure.
procedure	Describes roughly the process of this plan.

Associations

goalsEntityReference	The set of goals which this plan aims to achieve.
actionsEntityReference	The actions performed by this plan.
perceptsEntityReference	The percepts handled by this plan.
triggersEntityReferencePercept	The percepts triggering this plan.
writtenData	The data written by this plan.
readData	The data read by this plan.
incomingMessagesEntityReference	The messages received by this plan.
outgoingMessagesEntityReference	The messages sent by this plan.
triggersEntityReferenceMessage	The messages triggering this plan.
agent	Agents owning this plan.
capability	Capabilities owning this plan.

4.4 Consistency constraints

In addition to the metamodel, another important input to our change propagation framework is a set of consistency constraints. These constraints are usually derived based on well-formedness, coherence between diagrams and best practice guidelines[11]. With respect to Prometheus, the consistency constraints aim to ensure well-formedness and coherence among analysis overview diagrams, scenarios, role diagrams, data coupling diagram, agent role coupling diagram, system overview diagram, agent overview diagrams, and capability diagrams. In addition, the constraints make sure that best practice guidelines (such as those proposed in [Padgham and Winikoff, 2004]) are followed. These guidelines include constraints such as messages must be both sent and received, or data must be both used and produced.

The set of consistency constraints for Prometheus that we have used come from three

[11]It is noted that there are also constraints corresponding to the cardinalities (of associations) in the metamodel. Such constraints are also taken into account during the constraint repair process of our change propagation framework. In fact, our tool support (discussed in chapter 8) is able to extract information relating to cardinalities in the input metamodel and automatically create corresponding constraints for them.

different sources: the methodology documentation (mainly from [Padgham and Winikoff, 2004]), its tool support (i.e. Prometheus Design Tool), and our own contribution including consultation with the methodology's authors. More specifically, all the following constraints for roles, agents (except the last one), percepts, messages, and data are explicitly mentioned in [Padgham and Winikoff, 2004]. The constraint for Pelement is from [Winikoff, 2007]. Constraints 2-9 for plans are not explicitly discussed in the methodology's documentation but are enforced by the Prometheus Design Tool. The remaining constraints (all the constraints for steps) are derived based on our experience of using Prometheus plus consultation with the methodology's authors. We have used the Object Constrain Language (OCL) to formally express those constraints.

We now present the consistency constraints for entities in the Prometheus metamodel. For each constraint, there is an informal description and an equivalent OCL expression. It is noted that the context of an OCL constraint is implicitly the entity being described. For example, the context of the first constraint of the Role (below) is the entity Role in the metamodel.

4.4.1 Role

The relationship between roles and agents is constrained to be transitive: if a role is played by an agent, and the role reads some data, then the agent is deemed to also read the data. Most of the following constraints specify this transitive relationship.

1 Any data read by a role must also be read by all the agents playing this role.

```
self.readData→forAll( data : Data |
        self.agent→forAll(a : Agent | data.readByEntityReference_Agent→includes(a))
```

2 Any data written by a role must also be written by all the agents playing this role.

```
self.writtenData→forAll( data : Data |
        self.agent→forAll(a : Agent | data.writtenByEntityReference_Agent→includes(a))
```

3 Any action performed by a role must also be performed by all the agents which play this role.

```
self.actionsEntityReference→forAll(action : Action |
        self.agent→forAll(a : Agent | a.actionsEntityReference→includes(action)))
```

4 Any percept handled by a role must also be responded to by all the agents which play this role.

```
self.perceptsEntityReference→forAll(percept : Percept |
        self.agent→forAll(a : Agent | a.perceptsEntityReference→includes(percept)))
```

5 Any goal achieved by a role must also be accomplished by all the agents which play this role.

```
self.goalsEntityReference→forAll(goal : Goal |
        self.agent→forAll(a : Agent | a.goalsEntityReference→includes(goal)))
```

4.4.2 Agent

Constraints 1-5 are derived from the relationship between agents and roles. The rest of the constraints (except the last one) are derived from the completeness conditions between agents and plans, e.g. for each action performed by an agent, it is required to have at least one plan owned by the agent which performs this action[12]. Finally, the last constraint, i.e. constraint 16, is an example of a domain specific constraint. It describes a design guideline specific to the design of the weather alerting system.

1 Any percept handled by an agent must be handled by at least one of its roles.

```
self.perceptsEntityReference→forAll(percept : Percept |
        self.rolesEntityReference→exists(role : Role | role.perceptsEntityReference→includes(percept)))
```

2 Any action performed by an agent must be performed by at least one of its roles.

```
self.actionsEntityReference→forAll(action : Action |
        self.rolesEntityReference→exists(role : Role | role.actionsEntityReference→includes(action)))
```

3 Any goal achieved by an agent must be achieved by at least one of its roles.

```
self.goalsEntityReference→forAll(goal : Goal |
        self.rolesEntityReference→exists(role : Role | role.goalsEntityReference→includes(goal)))
```

4 Any data written by an agent must be written by at least one of its roles.

```
self.writtenData→forAll(data : Data |
        self.rolesEntityReference→exists(role : Role | role.writtenData→includes(data)))
```

5 Any data read by an agent must be read by at least one of its roles.

```
self.readData→forAll(data : Data |
        self.rolesEntityReference→exists(role : Role | role.readData→includes(data)))
```

[12]The action can be performed by a capability belonging to the agent. However, due to a similar constraint imposed on capabilities and the transitive relationship between agents, capabilities, and plans (refer to section 4.4.3), the capability needs to have at least one plan performing the action, and this plan is also ultimately owned by the agent.

6 For each action performed by an agent, there is at least one plan owned by the agent which performs this action.

```
self.actionsEntityReference→forAll(action : Action |
        self.includedPlansEntityReference→exists(plan : Plan | plan.actionsEntityReference→includes(action)))
```

7 For each percept handled by an agent, there is at least one plan owned by the agent which handles this percept.

```
self.perceptsEntityReference→forAll(percept : Percept |
        self.includedPlansEntityReference→exists(plan : Plan | plan.perceptsEntityReference→includes(percept)))
```

8 For each goal allocated to an agent, there is at least one plan owned by the agent which achieves this goal.

```
self.goalsEntityReference→forAll(goal : Goal |
        self.includedPlansEntityReference→exists(plan : Plan | plan.goalsEntityReference→includes(goal)))
```

9 For each message sent by an agent, there is at least one plan belonging to the agent that sends this message.

```
self.sentMessage→forAll(msg : Message |
        self.includedPlansEntityReference→exists(plan : Plan |
                                    plan.outgoingMessagesEntityReference→includes(msg)))
```

10 For all messages received by an agent, there at least one plan belonging to the agent that receives this message or is triggered by this message.

```
self.receivedMessage→forAll(msg : Message |
        self.includedPlansEntityReference→exists(plan : Plan |
                                    plan.incomingMessagesEntityReference→includes(msg)
                                or plan.triggersEntityReferenceMessage→includes(msg)))
```

11 For all messages posted internally within an agent there is at least one plan belonging to the agent that sends this message.

```
self.internalMessage→forAll(msg : Message |
        self.includedPlansEntityReference→exists(plan : Plan |
                                    plan.outgoingMessagesEntityReference→includes(msg)))
```

12 For all messages posted internally within an agent there is at least one plan belonging to the agent that receives or is triggered by this message.

```
self.internalMessage→forAll(msg : Message |
        self.includedPlansEntityReference→exists(plan : Plan |
                                    plan.incomingMessagesEntityReference→includes(msg)
                                or plan.triggersEntityReferenceMessage→includes(msg)))
```

13 For all data read by an agent, there is at least one plan belonging to the agent that reads this data.

```
self.readData→forAll(data : Data |
        self.includedPlansEntityReference→exists(plan : Plan | plan.readData→includes(data)))
```

14 For all data written by an agent, there is at least one plan belonging to the agent that writes this data.

```
self.writtenData→forAll(data : Data |
        self.includedPlansEntityReference→exists(plan : Plan | plan.writtenData→includes(data))
```

15 For all data owned by an agent (internal), there is at least one plan belonging to the agent that either reads/writes this data.

```
self.internalData→forAll(data : Data |
        self.includedPlansEntityReference→exists(plan : Plan |
                plan.writtenData→includes(data) or plan.readData→includes(data) ))
```

16 (Domain specific to the weather alerting system presented in section 4.1) The "GUI" agent is notified (of a particular change) only by agents that the "GUI" agent has subscribed to, i.e. for each message received by the agent "GUI", the agent which sends this message receives the "SubscribeChange" message from the "GUI" agent.

```
self.name = 'GUI' implies self.receivedMessage→forAll(msg : Message |
        msg.agentSender→forAll(a : Agent |
                a.receivedMessage→exists(m : Message |
                        m.name = 'SubscribeChange' and m.agentSender→includes(self))))
```

4.4.3 Capability

Since capabilities can contain sub(capabilities), the above constraints 6-15 that are applied to agents are also applied to capabilities. For instance, constraint 6 applying to a capability can be stated that for each action performed by a capability, there is at least one plan owned by the capability which performs this action. In addition, the relationships between agents, capabilities and plans are constrained to be transitive. For example, if capability C is owned by agent A and owns plan P (or capability C'), then agent A also owns plan P (or capability C'). The following constraints specify these transitive relationships.

1 A capability cannot own itself.

```
self.includedCapabilitiesEntityReference→excludes(self)
```

2 Any plan owned by a capability must also belong to all the agents owning the capability.

```
self.includedPlansEntityReference→forAll(pl : Plan |
        self.agent→forAll(a : Agent | a.includedPlansEntityReference→includes(pl)))
```

3 Any plan owned by a capability must also belong to all the capabilities owning the capability.

```
self.includedPlansEntityReference→forAll(pl : Plan |
        self.ownerCapability→forAll(cap : Capability | cap.includedPlansEntityReference→includes(pl)))
```

4 Any (sub-)capability owned by a (parent) capability must also belong to all the agents owning the (parent) capability.

```
self.includedCapabilitiesEntityReference→forAll(cap : Capability |
        self.agent→forAll(a : Agent | a.includedCapabilitiesEntityReference→includes(cap)))
```

5 Any (sub-)capability owned by a (parent) capability must also belong to all the capabilities owning the (parent) capability.

```
self.includedCapabilitiesEntityReference→forAll(cap : Capability |
        self.ownerCapability→forAll(ancCap : Capability |
                                    ancCap.includedCapabilitiesEntityReference→includes(cap)))
```

4.4.4 Percept

1 A percept must either trigger or be responded to by at least one plan.

```
self.plan→size() > 0 or self.triggeredPlan→size() > 0
```

4.4.5 Step

1 For each data written by a step, there exists a role which performs the step and also writes this data.

```
self.dataWrittenEntityReference→forAll(data : Data |
        self.rolesEntityReference→exists(r : Role | r.writtenData→includes(data)))
```

2 For each data read by a step, there exists a role which performs the step and also reads this data.

```
self.dataReadEntityReference→forAll(data : Data |
        self.rolesEntityReference→exists(r : Role | r.readData→includes(data)))
```

The following constraint is applied to PerceptStep only.

3 Among the roles performing a given PerceptStep, there exists one role that is assigned to the percept associated with the step.

self.rolesEntityReference→exists(r : Role | r.perceptsEntityReference→includes(self.stepEntityReference))

The following constraint is applied to ActionStep only.

4 Among the roles performing a given ActionStep, there exists one role that is assigned to the action associated with the step.

self.rolesEntityReference→exists(r : Role | r.actionsEntityReference→includes(self.stepEntityReference))

The following constraint is applied to GoalStep only.

5 Among the roles performing a given GoalStep, there exists one role that is assigned to the goal associated with the step.

self.rolesEntityReference→exists(r : Role | r.goalsEntityReference→includes(self.stepEntityReference))

4.4.6 Message

1 For each message, there should be at least one plan being triggered by it or receiving it.

self.planReceiver→size() > 0 or self.triggeredPlan→size() > 0

4.4.7 Pelement

1 Each protocol element (Pelement) must belong to exactly one protocol or one region.

(self.owner→size() = 1 and self.regionOwner→size() = 0)
xor (self.owner→size() = 0 and self.regionOwner→size() = 1)

4.4.8 Plan

1 A plan should have exactly one trigger (can be either a message or a percept).

(self.triggersEntityReferenceMessage→size() = 1 and self.triggersEntityReferencePercept→size() = 0)
xor
(self.triggersEntityReferenceMessage→size() = 0 and self.triggersEntityReferencePercept→size() = 1)

2 Any action performed by the plan should be performed by all the agents and capabilities owning the plan.

```
self.actionsEntityReference→forAll(action : Action |
        self.agent→forAll(agent : Agent | agent.actionsEntityReference→includes(action)))
        and self.capability→forAll(cap : Capability | cap.actionsEntityReference→includes(action)))
```

3 Any data written by a plan should be either owned by the plan's agents (in which case it should be written by capabilities that own the plan), or written by the agents (in which case it should also be written by capabilities that own the plan), or owned by the plan's capabilities.

```
self.writtenData→forAll(data : Data |
   self.agent→forAll(agent : Agent |
      (agent.internalData→includes(data)
         and self.capability→forAll(cap : Capability | cap.writtenData→includes(data)))
      or (agent.writtenData→includes(data)
         and self.capability→forAll(cap : Capability | cap.writtenData→includes(data))))
   or
   self.capability→forAll(cap : Capability | cap.internalData→includes(data)))
```

4 Any data read by a plan should be either owned by the plan's agents (in which case it should be read by capabilities that own the plan), or read by the agents (in which case it should also be read by capabilities that own the plan), or owned by the plan's capabilities.

```
self.readData→forAll(data : Data |
   self.agent→forAll(agent : Agent |
      (agent.internalData→includes(data)
         and self.capability→forAll(cap : Capability | cap.readData→includes(data)))
      or (agent.readData→includes(data)
         and self.capability→forAll(cap : Capability | cap.readData→includes(data))))
   or
   self.capability→forAll(cap : Capability | cap.internalData→includes(data)))
```

5 Any percept triggering the plan should be handled by all the agents and capabilities owning the plan.

```
self.triggersEntityReferencePercept→forAll(percept : Percept |
        self.agent→forAll(agent : Agent | agent.perceptsEntityReference→includes(percept)))
        and self.capability→forAll(cap : Capability | cap.perceptsEntityReference→includes(percept)))
```

6 Any percept handled by the plan should be handled by all the agents and capabilities owning the plan.

```
self.perceptsEntityReference→forAll(percept : Percept |
        self.agent→forAll(agent : Agent | agent.perceptsEntityReference→includes(percept)))
        and self.capability→forAll(cap : Capability | cap.perceptsEntityReference→includes(percept)))
```

7 Any message sent by a plan should be either owned by the plan's agents (in which case it should be sent by capabilities that own the plan), or sent by the agents (in which case it should also be sent by capabilities that own the plan), or owned by the plan's capabilities.

```
self.outgoingMessagesEntityReference→forAll(msg : Message |
  self.agent→forAll(agent : Agent |
    (agent.internalMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.sentMessage→includes(msg)))
    or (agent.sentMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.sentMessage→includes(msg))))
  or
  self.capability→forAll(cap : Capability | cap.internalMessage→includes(msg)))
```

8 Any message received by a plan should be either owned by the plan's agents (in which case it should be received by capabilities that own the plan), or received by the agents (in which case it should also be received by capabilities that own the plan), or owned by the plan's capabilities.

```
self.incomingMessagesEntityReference→forAll(msg : Message |
  self.agent→forAll(agent : Agent |
    (agent.internalMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.receivedMessage→includes(msg)))
    or (agent.receivedMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.receivedMessage→includes(msg))))
  or
  self.capability→forAll(cap : Capability | cap.internalMessage→includes(msg)))
```

9 Any message triggering a plan should be either owned by the plan's agents (in which case it should be received by capabilities that own the plan), or received by the agents (in which case it should also be received by capabilities that own the plan), or owned by the plan's capabilities.

```
self.triggersEntityReferenceMessage→forAll(msg : Message |
  self.agent→forAll(agent : Agent |
    (agent.internalMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.receivedMessage→includes(msg)))
    or (agent.receivedMessage→includes(msg)
      and self.capability→forAll(cap : Capability | cap.receivedMessage→includes(msg))))
  or
  self.capability→forAll(cap : Capability | cap.internalMessage→includes(msg)))
```

4.4.9 Data

1 Any data must be read or written or owned by at least one agent.

self.agentOwner→size() > 0 *or* self.readByEntityReference_Agent→size() > 0
or self.writtenByEntityReference_Agent→size() > 0

2 Any data must be read or written by at least one plan.

self.planReader→size() > 0 *or* self.planWriter→size() > 0

4.5 Example

In this section, we provide a simple example that illustrates how the constraints defined are used by our framework to perform change propagation in Prometheus. The example that we use is the design of a weather alerting system that has been shown in the previous sections.

In this example, we focus on a particular agent, the "Alerter" agent. As shown in figure 4.5 (on page 73), within the existing system this agent plays two roles: "Manage Subscription" and "Filter Alerts". Figure 4.4 (on page 72) shows that the role "Manage Subscription" is assigned to achieve goal "Subscribe Registration" and role "Filter Alerts" achieves goals "Determine Frequency" and "Package Alerts". As can be seen in figure 4.17, the two roles have access to only one data, i.e. "SubscriptionsStore".

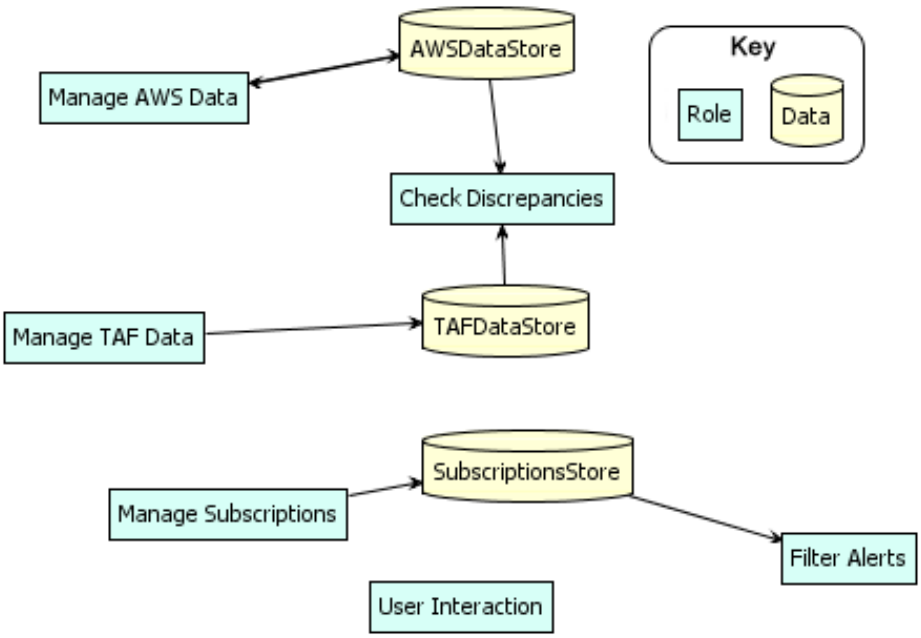


Figure 4.17: Data coupling diagram for a weather alerting system

Figure 4.18 depicts the internals of the Alerter agent. It currently has two plans, "SendAlert-

ToSubscribedGUIs" and "HandleAlertSubscription". The former is for sending new alerts to subscribed "GUI" agents, and the latter is for handling subscriptions, which are stored in the "SubscriptionsStore" data.

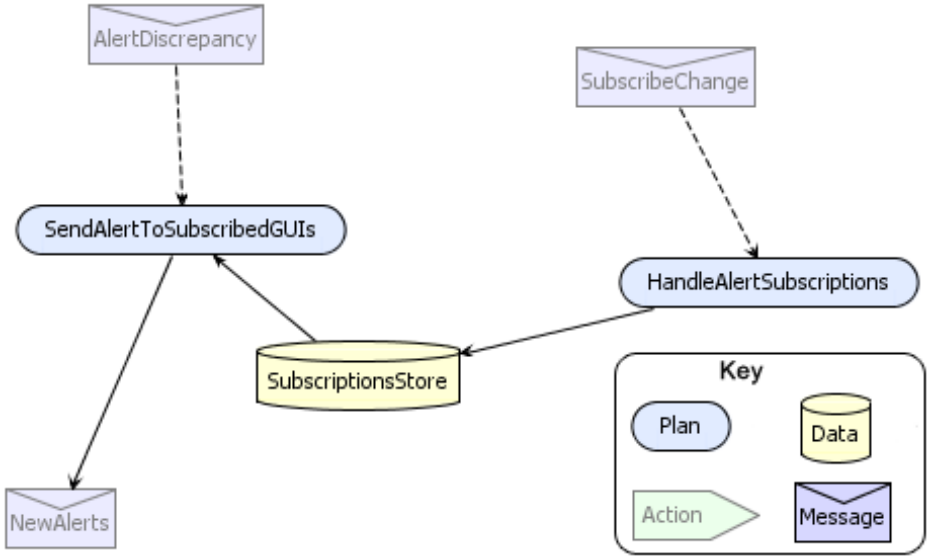


Figure 4.18: Agent overview diagram for the "Alerter" agent

Let us suppose that as a means of tracking alerts generated and also for verification purposes, a new requirement is that all alerting messages sent to the "GUI" agents of the system should be recorded in a log. The "Alerter" agent is responsible for handling alerts and sending them to the registered "GUI" agents as well as managing the alert subscriptions. Hence, one way of implementing this change is making the "Alerter" agent be responsible for handling alert logging.

Since we need a new data store to log the details of alerts sent, it is assumed that the designer would perform the following primary changes:

1. Create a new data called "AlertLog" (equivalent to creating a new instance of the Data class in the metamodel and naming it "AlertLog").

2. Making the "Alerter" agent write to the new data (equivalent to connecting agent "Alerter" to data "AlertLog" with the relationship *writtenByEntityReference_Agent* – *writtenData* in the metamodel – see figure 4.12 on page 87).

Assume that after making the above changes, the designer uses our framework to find out

if those changes cause any inconsistency in the design model and if so, what further changes are needed to resolve them.

The constraint checking component of our framework evaluates all the constraints presented in section 4.4. It identifies constraint 4[13] and constraint 14[14] (in the context of an Agent), and constraint 2[15] (in the context of a Data) as being violated.

- C4("Alerter"): constraint C4 (in the Agent context) evaluated on agent "Alerter".
- C14("Alerter"): constraint C14 (in the Agent context) evaluated on agent "Alerter".
- C2("AlertLog"): constraint C2 (in the Data context) evaluated on data "AlertLog".

In order to fix constraint *C4("Alerter")*, our repair plan generator generates the following repair options. Note that in chapter 6, we will discuss in detail how these repair options are generated.

1. Making either existing roles be played by the "Alerter" agent (i.e. "Manage Subscription" or "Filter Alerts") write to the "AlertLog" data.
2. Making one of the other existing roles (but not "Manage Subscription" or "Filter Alerts") be played by the "Alerter" agent, and write to the "AlertLog" data.
3. Creating a new role and making it be played by the "Alerter" agent, and write to the "AlertLog" data.
4. Making the "Alerter" agent not write to the "AlertLog" data.

The first three repair options do not violate any other constraints. However, the last repair option not only undos the previous primary change but also breaks the constraint 1[16] in the context of a Data. Therefore, if it gets chosen, further changes are needed and may also result in a cycle.

With regard to constraint *C14("Alerter")*, our repair plan generator gives several repair options, which involve either the creation of new plans or new capabilities or using existing plans to access the newly created data. For instance, the obvious and easy fix is to make

[13] Any data written by an agent must be written by at least one of its roles.
[14] For all data written by an agent, there is at least one plan belonging to the agent that writes this data.
[15] Any data must be read or written by at least one plan.
[16] Any data must be read or written or owned by at least one agent.

one of the plans belong to agent "Alerter" (i.e. "SendAlertToSubscribedGUIs" or "HandleAlertSubscription") write to the "AlertLog" data. However, this option seems to degrade the coherence of those existing plan in agent "Alerter" by adding in an unrelated task. As a result, the user may chose the repair option involving the creation of a new plan.

Similarly with regard to constraint *C2("AlertLog")*, there are also various repair options that can resolve its violation. One of the ways of fixing this constraint is to make an existing plan read or write to the "AlertLog" data. However, doing so would violate constraint 3[17] or 4[18] (in the context of a Plan) because an agent which owns the plan needs to either read or write to the data as well. There is an exception (i.e. no further constraint violations) if either plan "SendAlertToSubscribedGUIs" or "HandleAlertSubscription" is used, since the agent owning it ("Alerter" agent) already writes to the "AlertLog" data). This option also resolves the violation of constraint 14 (in the context of an Agent).

Our framework uses the above reasoning process to work out the best (i.e. cheapest) repair options and propose them to the user. For instance, in this example one of the repair options that is proposed by our framework is to make the role "Manage Subscription") write to the "AlertLog" data (fixing constraint 4 in the context of an Agent) and make the plan "SendAlertToSubscribedGUIs" write to the "AlertLog" data (fixing constraint 14 in the context of an Agent and constraint 2 in the context of a Data).

4.6 Chapter summary

In this chapter we have briefly described the Prometheus methodology, which is the subject of our major case study to demonstrate the applicability of our change propagation framework. We have provided an overview of the major phases and artefacts within Prometheus and refer the reader to other technical documentation (e.g. [Padgham and Winikoff, 2004]) for more details. The key content of this chapter, which also highlights one of our contributions, is the development of a metamodel and a set of consistency constraints for Prometheus. We first explained where a Prometheus metamodel would be placed in a hierarchy of metamodels like the popular MDA's four-layer metamodel hierarchy. We then presented our proposal

[17] Any data written by a plan should be either owned by the plan's agents (in which case it should be written by capabilities that own the plan), or written by the agents (in which case it should also be written by capabilities that own the plan), or owned by the plan's capabilities.

[18] Any data read by a plan should be either owned by the plan's agents (in which case it should be read by capabilities that own the plan), or read by the agents (in which case it should also be read by capabilities that own the plan), or owned by the plan's capabilities.

of a Prometheus metamodel with a detailed description. We also described a number of consistency constraints that we have identified based on conditions such as well-formedness of the models, coherence between diagrams, and best practice guidelines.

In order to illustrate how our framework uses the defined metamodel and consistency constraints to perform change propagation in Prometheus models, we gave a simple example based on a weather alerting system which was designed using the Prometheus methodology. Nonetheless, the example only shows the reasoning process of the framework at a high level. More details of the framework are presented in the chapters ahead. In the next chapter, we will, however, investigate the application of our framework to UML, a widely used modelling language.

Chapter 5

Case Study II: UML

Our framework is generally applicable to a range of methodologies and design types. In the previous chapter, we have discussed a case study in which our framework was applied to Prometheus, an agent-oriented methodology. In this chapter, we present a small case study with the aim of showing how our framework can support change propagation in UML object-oriented design models. We first give a brief overview of UML in section 5.1 and describe a small excerpt of the UML metamodel and some examples of consistency constraints that are commonly used in UML practice (section 5.2). We then discuss a small example to illustrate how our framework deals with changes in a UML design of an existing system.

5.1 Overview of UML

The Unified Modelling Language (UML) has its roots in the unification of various object-oriented modelling languages emerging in the early 1990s. In 1997, UML was adopted by the Object Management Group (OMG) as an official OMG standard. Since then, UML has become the state-of-the-art and widely-used modelling language, especially for object-oriented software development. UML has undergone various versions and the latest version 2.0 has recently been released. However, UML 2.0 is not commonly supported in industry, in part, because of legacy models and tools [Egyed, 2007]. In this case study, we apply our framework to UML 1.4.2 [Object Management Group, 2005], which was adopted as an ISO standard. However, our ideas and results can be applied to other versions of UML.

UML has a set of diagrams which provide multiple perspectives of the system under development. Figure 5.1 describes major diagram types in UML and their purposes. A *use*

case diagram describes the functionalities of a system in terms of how users interact with the system. A *class diagram* is the backbone of UML which describes the static structure of a system in terms of the types of objects (i.e. classes) and relationships between them. An *object diagram* is closely linked to class diagrams, which describes the static structure of an instance of the system at a particular time. The dynamic interactions between objects are modelled using sequence diagrams and communication diagrams. *Sequence diagrams* depict interactions between objects in terms of an exchange of messages over time. Sequence diagrams emphasise the sequence of messages in an interaction. Although *communication diagrams* also capture a series of sequenced messages, their main emphasis is on the communication paths between different interacting objects.

Diagram	**Behaviour or Structure**	**Purpose**
Use case	Behaviour/ Structure	How users interact with a system
Class	Structure	Classes and static relationships between them
Object	Structure	An instance of a class diagram, showing a snapshot of the detailed state of a system at a point in time
Sequence	Behaviour	Possible interactions between objects with the focus on sequence
Collaboration	Behaviour	Possible interactions between objects with the focus on links
Statechart	Behaviour	Possible sequences of states and actions through which an object can proceed during its lifetime
Activity	Behaviour	Dynamic nature of a system in terms of the flow of control from activity to activity
Component	Structure	Aspects of physical implementation in terms of structure and connections of components
Deployment	Structure	Physical resources in a system, including nodes, components, and connections.

Figure 5.1: UML diagram types

Statechart diagrams are used to describe the dynamic behaviour of a system. In particular, a statechart diagram defines the possible states an object can proceed through during its lifetime, and the different transitions from one state to another. Similar to statechart diagrams, *activity diagrams* also illustrate the dynamic nature of a system. In fact, an activity diagram is a special case of a state diagram. However, activity diagrams are typically used to describe workflow or business processes and procedural logic. Because an activity diagram is a special kind of statechart diagram, it uses some of the same modelling conventions. Aspects

of physical implementation of a system are modelled in two different types of implementation diagrams: component diagrams and deployment diagrams. *Component diagrams* depict the structure of components whereas *deployment diagrams* illustrate the structure of the nodes on which the components are deployed.

Those UML diagrams represent different views of elements of a common UML metamodel that we describe in the next section.

5.2 UML metamodel

UML is defined using a metamodelling approach. Figure 5.2 depicts a four-layer metamodel hierarchy in which the UML metamodel is built. At M0 is the actual system with run-time instances, i.e. object instances. At the M1 layer is the model of a system using UML. The UML metamodel is situated at the M2 layer, which contains instances of MOF model elements (M3).

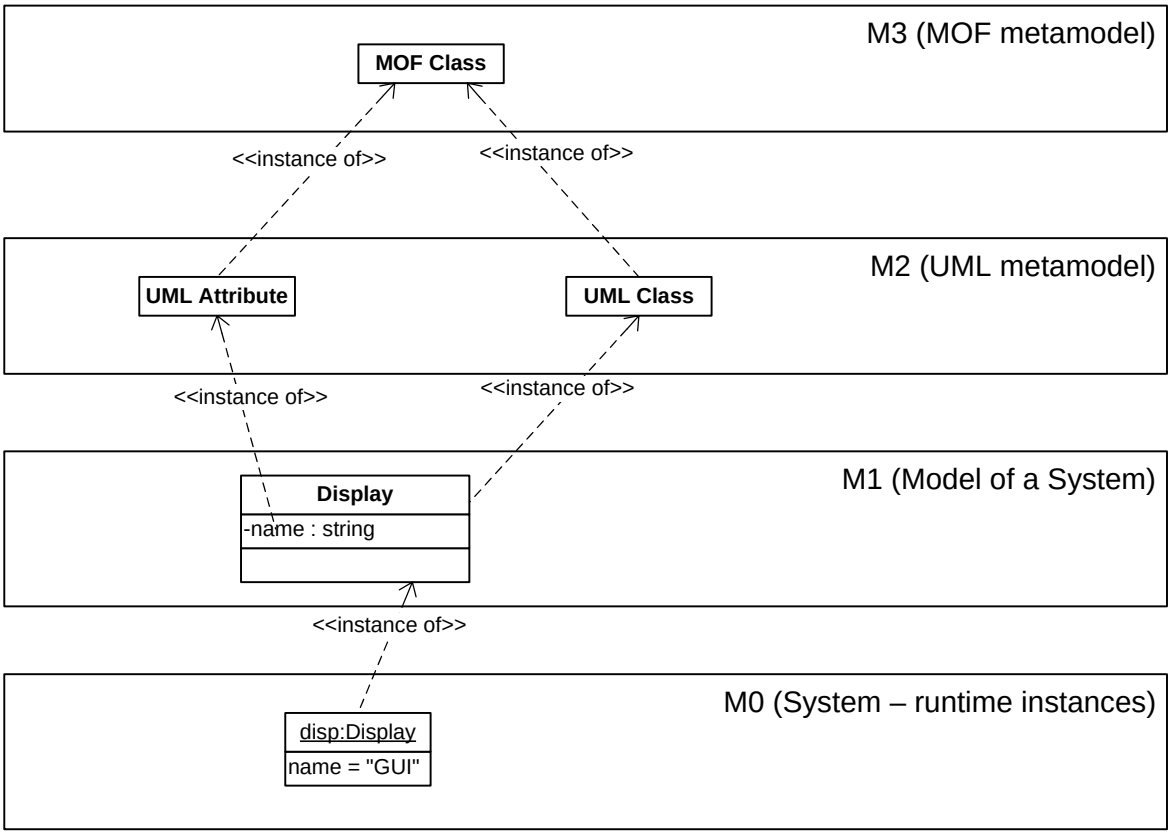


Figure 5.2: The four-layer metamodel hierarchy

The UML metamodel contains the concrete syntax, abstract syntax and the semantics of UML. The UML metamodel is also equipped with well-formedness rules that are described in the Object Constraint Language (OCL). OCL is also used to express the semantics of UML, and heuristics and even best practices in UML. The UML metamodel and its accompanying well-formedness conditions are very large and the full details are provided in [Object Management Group, 2005]. Figures 5.3[1] and 5.4 show small excerpts of the UML metamodel. Figure 5.3 depicts relationships between major elements in a class diagram: *Class*, *Association*, *AssociationEnd* and *Operation*, and between two major elements in a sequence diagram: *ClassifierRole* (usually referred to as objects in a sequence diagram) and *Message*. According to the UML metamodel, a *Class* has many *AssociationEnds*, and at least two of those are needed to form an *Association*. In addition, a *Class* can own multiple *Operations* and can be a base of several *ClassifierRoles*, which can send or receive messages.

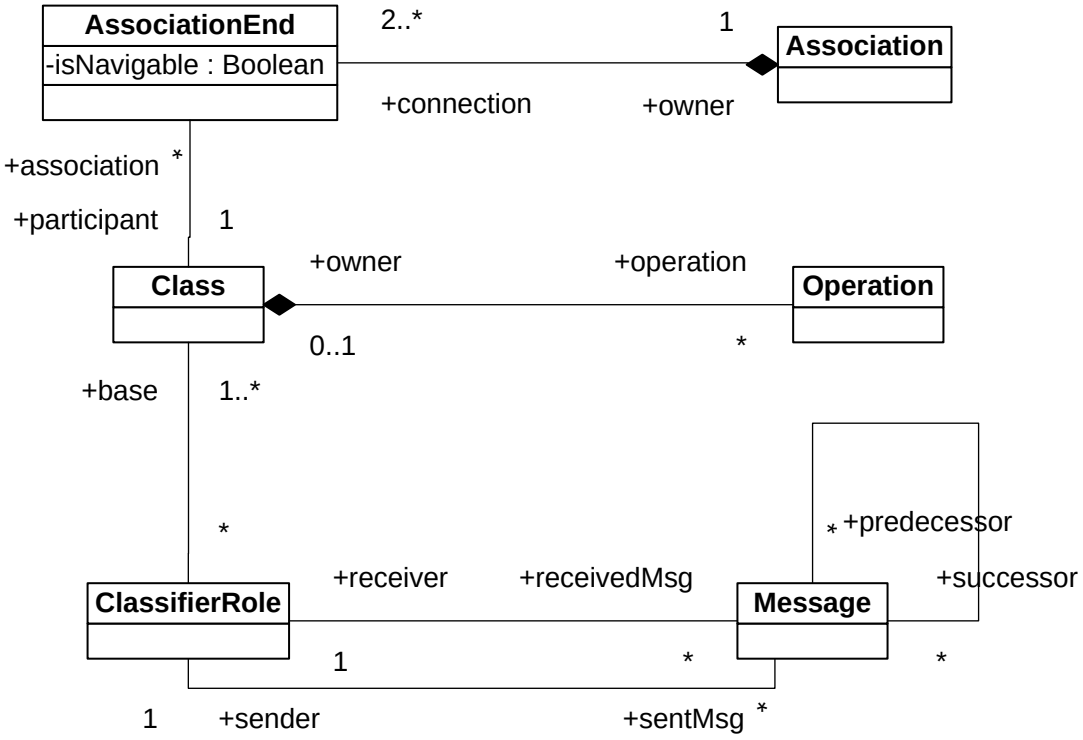


Figure 5.3: An excerpt of UML metamodel concerning Class*,* Operation*,* ClassifierRole*, and* Message

[1]It is noted that although in the metamodel a ClassifierRole can have more than one base Class, this cardinality is usually only one in practice.

Figure 5.4 depicts the *key* elements of a state machine[2] (alternatively called a *statechart diagram*). The behaviour of each *Class* can be specified by multiple *StateMachines* (although one is sufficient for most purposes). A *StateMachine* contains a top-level *State* and a set of *Transitions*. All remaining states are transitively owned by a state machine through its top state and the state containment hierarchy. A State can be either a CompositeState (that contains other states), a SimpleState or a FinalState. Each *Transition* has a source *State* and a target *State* that is reached when the transition is taken. A transition can have at most one trigger, which is the *Event* that fires the transition. Each state has transitions departing from it and entering it.

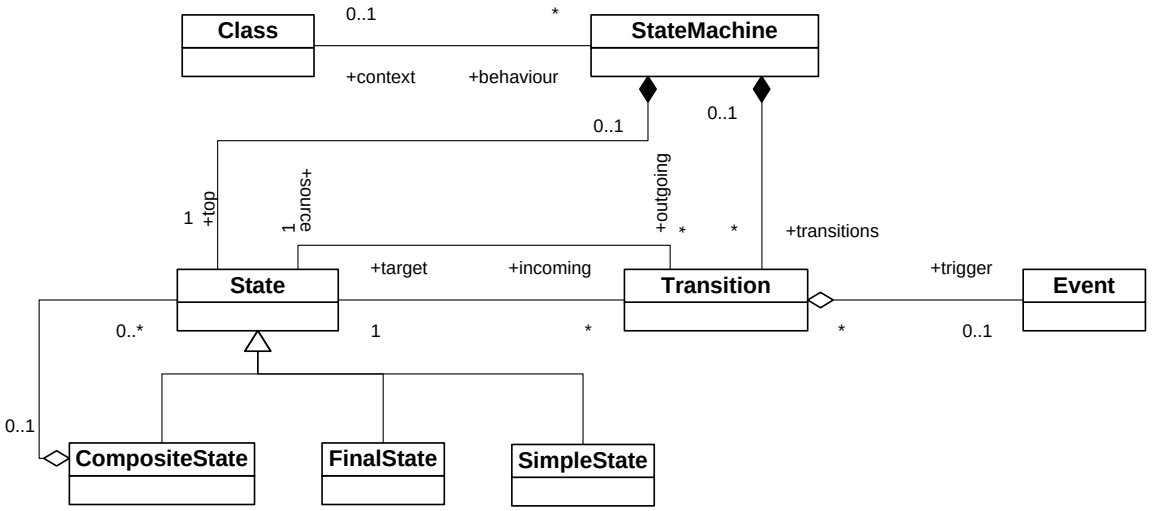


Figure 5.4: An excerpt of UML metamodel concerning Class, StateMachine, State, Transition

In this case study we consider only a fragment of the UML with the focus on three major diagrams: class, sequence and statechart diagrams. Class diagrams are the most important structural diagrams which capture the fundamental concepts of object-oriented development: classes and their relationships. Sequence and statechart diagrams also play an important part in modelling in UML in which they capture the behavioural aspect. There is a great deal of overlap between the three types of diagram. Model elements defined in class diagrams are used in sequence and statechart diagrams. Sequence diagrams and statechart diagrams are complementary in the way that the former models interactions between objects, whilst the latter depicts the behaviour of a single object. The three diagram types also cover a

[2]For the sake of simplicity we abstract away other entities such as StateVertex, PseudoState, Guard, SubState, etc.

large range of modelling elements in UML. In addition, they are the diagram types most commonly described and studied in literature. Other types of diagram such as use case and activity diagrams are also interesting. However, due to time restrictions we were not able to investigate these types of diagram.

Consistency constraints for UML specify conditions that an UML model must obey for it to be considered a valid UML model, e.g. syntactic well-formedness and coherence between different diagrams. Below are four such consistency constraints[3] on how UML class, sequence and statechart diagrams relate to each other. It is noted that constraints 1 and 3 are standard UML well-formedness constraints [Object Management Group, 2005], whilst constraints 2 and 4 are not, but they are examples of coherence constraints between two diagrams [Egyed, 2006].

1 The name of a message (in a sequence diagram) must match an operation in its receiver's class (in a class diagram). (**c1**)

Context Message inv c1:
self.receiver.base.operation→exists(op : Operation | op.name = self.name)

2 The sequence of incoming messages in an object of a sequence diagram must match the allowed behaviour of the statechart diagram of the object's class. (**c2**)

It is rather complicated to express this constraint in OCL. Similarly to [Egyed, 2007], we provide here a definition sketch[4]. The idea is that for a given message, we find a state transition that matches the name of the message. This constraint holds for the message only if for a given sequence of messages in the sequence diagram (starting from the first message), there exists a sequence of state transitions from the first transition which matches the message sequence.
startingPoints = find state transitions equal to the first message name
startingPoints→exists(object sequence = reachable sequence from startingPoint in the statechart)

3 The message calling direction (in a sequence diagram) must match the class association (in a class diagram). (**c3**)

Context Message inv c3:
self.receiver.base.inAssociations.intersection(self.sender.base.outAssociations)→notEmpty()

where *inAssociations* and *outAssociations* are additional operations applied to a Class. *inAssociations* is the set of associations (of a given class) that have directions pointing to the class. Meanwhile, *outAssociations* is the set of associations (of a given class) that have directions pointing away from the class.
Context Class:

[3]It is noted that we extensively use the shorthand of the *collect* OCL operation here. For instance, in the first constraint *self.receiver.base* refers to the set of base classes of the *self*'s receiver and *self.receiver.base.operation* results in the set of operations of all those base classes.

[4]As a result, the checking of this constraint is currently manually done, not by the tool.

```
inAssociations : set(Association);
inAssociations = self.association.owner→select(a |
                    a.connection→select(ae | ae.participant = self and ae.isNavigable).size = 1)
outAssociations : set(Association);
outAssociations = self.association.owner→select(a |
                    a.connection→select(ae | ae.participant ⟨⟩ self and ae.isNavigable).size = 1)
```

4 The name of an event (in a statechart diagram of an object) must match an operation in the corresponding class. (**c4**)

Context Event inv c4:
self.transition.statemachine.context→exists(op : Operation | op.name = self.name)

However, we consider here only two constraints $c1$ and $c2$ for the purpose of comparing with the work in [Egyed, 2007], which also aims to fix inconsistencies in UML design models. He used the two constraints $c1$ and $c2$ to illustrate how his approach works. These two constraints form what we call a *repair scope*, which is a group of constraints that are considered together in the process of fixing constraint violations. Further discussion of a repair scope will be presented in chapter 7.

5.3 Case study application

Having provided a brief snapshot of key elements of the UML metamodel and some examples of consistency constraints, we now discuss a simple application which illustrates how our framework uses the UML metamodel and consistency constraints to propagate changes. We first describe the initial system and a simple change. We then explain how each process in our framework is executed to deal with the change.

5.3.1 Initial system

Our case study is a design of a real, albeit simplified, video on demand (VOD) system [Dohyung, 1999]. We use the simplified version that is presented in [Egyed, 2007] due to the fact that we want to compare his work with ours. The simplified VOD system allows a user to select a movie to play. The user is also able to play, pause and resume the movie.

The class diagram (see figure 5.5) represents the structure of the initial VOD system. There are three classes: “Display” for visualizing movie streams and receiving user inputs, “Streamer” for downloading and decoding movies, and “Server” for providing data. The “Display” class has four operations: “select()” for choosing a movie, “stream()” for playing

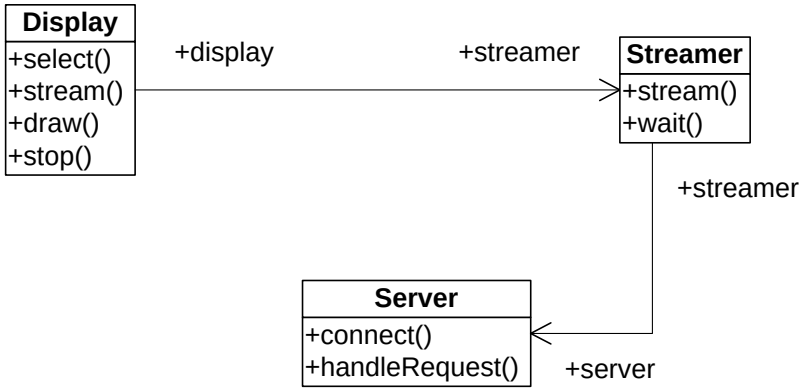


Figure 5.5: Class diagram for the VOD system (adopted from [Egyed, 2007])

and retrieving the movie steams, "draw()" for rendering the received movie stream, and "stop()" for halting the movie being played. The "Streamer" class has only two operations: "stream()" for streaming the movie data received from the server, and "wait()" for halting the streaming process. Finally, the "Server" has a "connect()" operation that is called by clients (e.g. the Streamer) and a "handleRequest()" to deal with requests from clients.

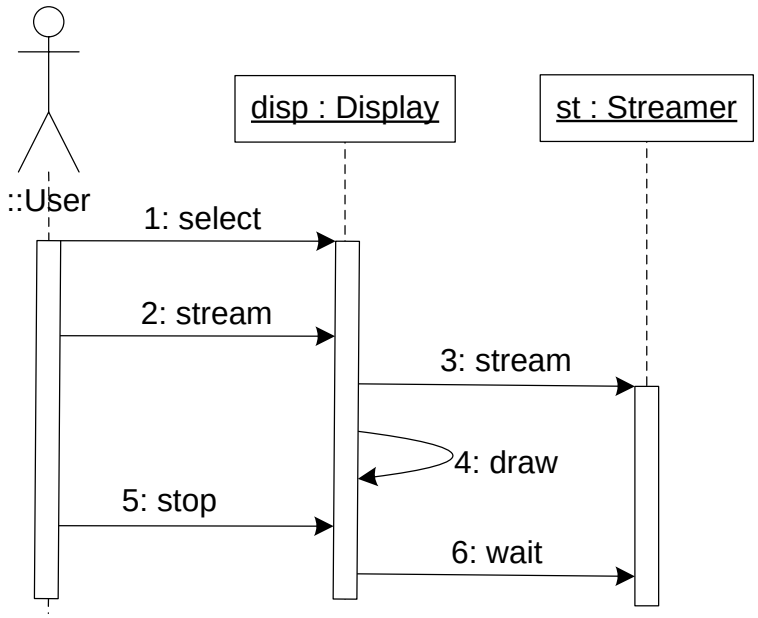


Figure 5.6: A sequence diagram for instances of classes Display and Streamer (adopted from [Egyed, 2007])

The sequence diagram (see figure 5.6) depicts a typical scenario of interactions between

the user, a Display object ("disp") and a Streamer object ("st")[5]. The user selects a movie that she wants to see (message 1). She then starts playing the selected movie - in the sequence diagram the user sends a message "stream" to the Display (message 2). The Display then retrieves the movie stream from the Streamer (message 3) and renders the movie (message 4). When the user wants to stop viewing the movie (message 5), the Display notifies the Streamer to stop streaming (message 6).

The two statechart diagrams (see figure 5.7) describe the behaviour of the two classes: "Display" and "Streamer". As can be seen, the behaviour of the "Streamer" class simply changes between the waiting and the streaming states depending on whether it is triggered by the "wait" or "stream" event. Meanwhile, the behaviour of the "Display" class ranges over three different states: "Idle", "Ready" and "Playing".

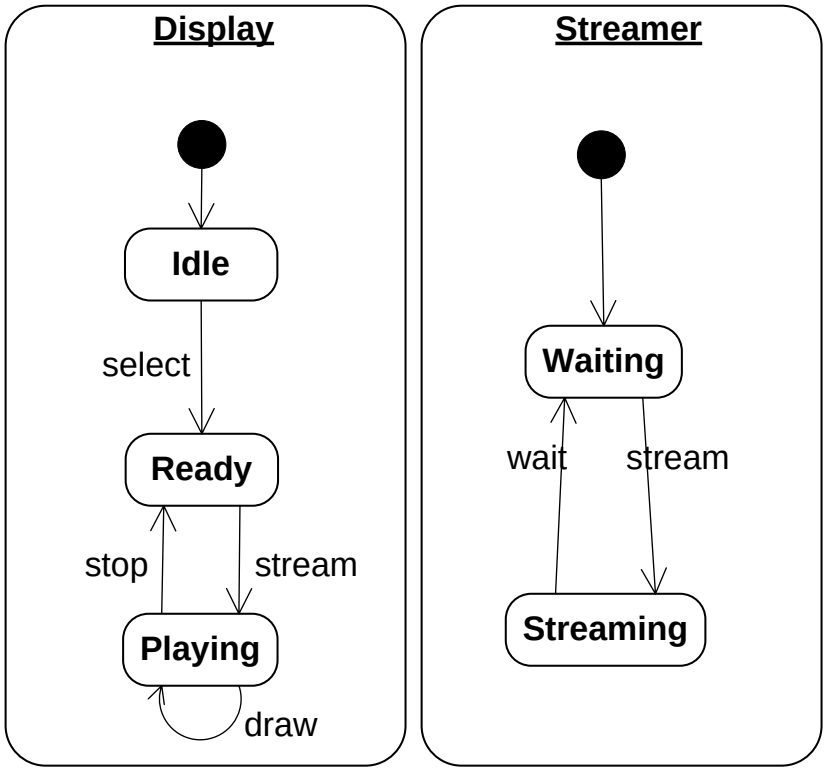


Figure 5.7: Statechart diagrams for classes Display and Streamer (adopted from [Egyed, 2007])

[5]A Server object is also involved in these interactions but we do not show it here.

5.3.2 A proposed change

In the current design, the Display and Streamer classes have two different methods with the same name "stream". In order to avoid the confusing dual use of the term "stream", the designer makes the following *primary* changes. It is emphasized that these changes are proposed by Egyed [2007] and intentionally include design errors for the purpose of illustrating how undesirable inconsistencies are identified and resolved.

A1: Renaming the method "stream()" of class Display to "play()".

A2: Renaming the message "3:stream" to "play" in the sequence diagram.

A3: Renaming the state transition named "stream" to "play" in the Display's statechart.

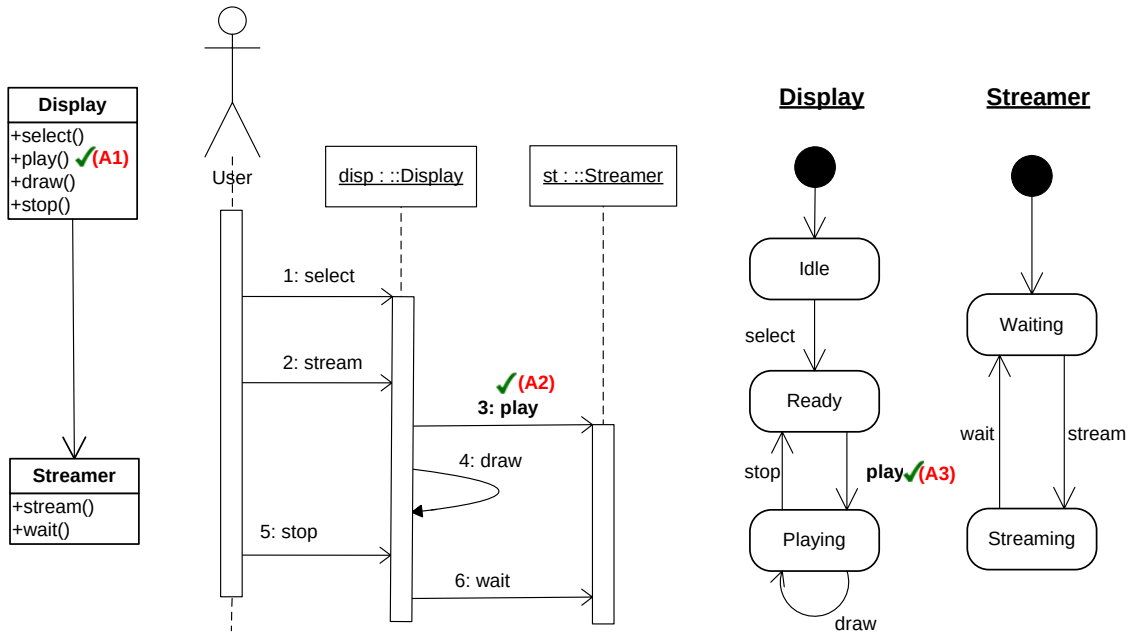


Figure 5.8: Design of the VOD system after primary changes are made (adopted from [Egyed, 2007])

Figure 5.8 shows how the above changes are made on the initial UML model (each change is marked with the ✓ sign). We now consider how our framework performs change propagation by restoring consistency in the design.

5.3.3 Change propagation process

The overall process of change propagation in our framework (described in detail in chapter 3) is summarized as follows.

1. At design time the repair plans are automatically generated from the constraints and metamodel
2. When the change propagation process starts, we check whether the constraints hold in the design model.
3. We use the repair plans to generate plan instances (i.e. repair options) for the violated constraints.
4. We calculate the cost of the different repair plan instances.
5. We select a repair plan instance (possibly by picking the single cheapest, if it exists, or by asking the user).
6. The selected repair plan instance is executed, and it updates the application design model.

We now describe how each of the steps in this process is applied to our VOD example.

Generate repair plan types

Let us consider the first constraint, denoted as $C1(self)$, that we have defined above. By analysing its definition[6], we are able to identify several ways of fixing $C1(self)$:

- Take an operation op in $self.receiver.base.operation$, and make $op.name = self.name$ true (which can be achieved by either renaming op or renaming $self$).
- Take an existing operation op which does not belong to $self.receiver.base.operation$, add op to $self.receiver.base.operation$, and make $op.name = self.name$ true.
- Create a new operation op, add op to $self.receiver.base.operation$, and make $op.name = self.name$ true.

[6]In chapter 6, we present a translation schema that takes OCL constraints as inputs, and generates a complete set of repair plans for them.

The above options are written in our repair plan syntax as follows[7].

Plan P 1. *c1True(self) : op ∈ self.receiver.base.operation ← !c1'True(op, self)*

Plan P 2. *c1True(self) : op ∈ Set(Operation) ∧ op ∉ self.receiver.base.operation ← !(Add op to self.receiver.base.operation) ; !c1'True(op, self)*

Plan P 3. *c1True(self) ← Create an operation op ; !(Add op to self.receiver.base.operation) ; !c1'True(op, self)*

In the body of plans P1, P2 and P3, an event $c1'True(op, self)$ is posted. This event corresponds to making $op.name = self.name$ true, which is handled by the following plans.

Plan P 4. *c1'True(op, self) : self.name ≠ op.name ∧ self.name ≠ null ← Rename op to self.name*

Plan P 5. *c1'True(op, self) : self.name ≠ op.name ∧ op.name ≠ null ← Rename self to op.name*

Plan P 6. *c1'True(op, self) : self.name = op.name ∨ c1(self) ← true*[8]

There is another event, *Add op to self.receiver.base.operation*, that is posted within the body of plans P2 and P3. Adding the operation *op* to the set of operations *self.receiver.base.operation* can be achieved in several ways, including[9] making *op* an operation of the class *self.receiver.base*, or changing *self.receiver.base* to an existing class that owns the operation *op*. These are expressed in terms of the following repair plans.

Plan P 7. *Add op to self.receiver.base.operation ← Connect self.receiver.base with op*

Plan P8. *Add op to self.receiver.base.operation : op ∈ x.operation ← !(Change self.receiver.base to x)*

Similarly, changing *self.receiver.base* to an existing class x that owns the operation *op* can be achieved in different ways and is consequently represented as an event. The plans that are able to handle this event include making x be the base of *self.receiver* (plan 9) or making the receiver of message *self* be an object that is an instance of class x (plan 10).

[7]It is noted that for each repair plan, there is an implicit context condition that the constraint and the parent constraints that it is trying to fix are violated.

[8]It means that this plan does nothing.

[9]Other options are less reasonable, e.g. creating a new class, adding *op* to be one of the new class's operations, and making the receiver's class of message *self* to be the new class.

Plan P9. *Change self.receiver.base to x* ← *Connect self.receiver to x*

Plan P10. *Change self.receiver.base to x : o* ∈ *Set*(*ClassifierRole*) ∧ *o.base* = *x* ← *Connect self to o*

Figure 5.10 summarises the repair plans (and subplans) for constraint C1. Using a similar approach we can derive repair plans for constraint C2.

C1	The name of a message (in a sequence diagram) must match an operation in its receiver's class (in a class diagram).
C2	The sequence of incoming messages in an object of a sequence diagram must match the allowed behaviour of the statechart diagram of the object's class.

Figure 5.9: Example consistency constraints

Check constraints

After the designer completes making the primary changes on the design of the initial system, our framework is in place to perform change propagation. The first step is checking constraints, which involves the instantiation of pre-defined constraints. For instance, the two constraints in our repair scope (see figure 5.9) are instantiated with respect to each instance of the constraints' context. For example, there are 6 instances of the first constraint, each corresponding to a message in the sequence diagram. Each constraint instance is evaluated to check for violation. For example, with respect to the constraint instance $C1(\text{“}2 : stream\text{”})$ the evaluation first computes *self.receiver.base.operation* where *self.receiver* is the object “disp” (this object is on the receiving end of the message as shown by the arrowhead), *receiver.base* is the class “Display” (object “disp” is an instance of class “Display”), and *base.operation* is {“*select*()”, “*play*()”, “*draw*()”, “*stop*()”} (the set of operations of the class “Display”). The evaluation then returns false because there does not exist any operation in the set *base.operation* that has the same name (i.e. *stream*) as message “2:stream”.

Following a similar approach, we identify the following constraints that are violated after the primary changes are made.

- C1(“2:stream”): constraint C1 evaluated on message “2:stream”.
- C1(“3:play”): constraint C1 evaluated on message “3:play”.
- C2(“disp”): constraint C2 evaluated on object “disp”.

- C2("st"): constraint C2 evaluated on object "st".

P1	c1True(self) : op ∈ self.receiver.base.operation ← !c1'True(op, self)
P2	c1True(self) : op ∈ Set(Operation) ∧ op ∉ self.receiver.base.operation ← !(Add op to self.receiver.base.operation) ; !c1'True(op, self)
P3	c1True(self) ← Create an operation *op* ; !(Add op to self.receiver.base.operation) ; !c1'True(op, self)
P4	c1'True(op, self) : self.name ≠ op.name ∧ self.name ≠ null ← Rename *op* to *self.name*
P5	c1'True(op, self) : self.name ≠ op.name ∧ op.name ≠ null ← Rename *self* to *op.name*
P6	c1'True(op, self) : self.name = op.name ∨ c1(self) ← true
P7	Add op to self.receiver.base.operation ← Connect *self.receiver.base* with *op*
P8	Add op to self.receiver.base.operation : *op* ∈ *x.operation* ← !(Change *self.receiver.base* to *x*)
P9	Change *self.receiver.base* to *x* ← Connect *self.receiver* to *x*
P10	Change *self.receiver.base* to *x* : *o* ∈ *Set*(*ClassifierRole*) ∧ *o.base* = *x* ← Connect *self* to *o*

Figure 5.10: Example repair plans for constraint $C1$

Generate repair plan instances

After violated constraints are identified, the next step in our change propagation framework is generating plan instances for each of the violated constraints. It is important to note that each repair plan type can generate multiple (i.e. zero or more) plan instances, depending on its context condition. For instance, let us consider the repair plan instances for constraint $C1$("2 : *stream*") (where *self* = "2 : *stream*" and *self.name* = "*stream*") (refer to figure 5.10 for the repair plan types). Since *self.receiver.base.operation* = {"*select*()", "*play*()", "*draw*()", "*stop*()"}, repair plan P1 generates 4 plan instances (plans $P1_1$, $P1_2$, $P1_3$, and $P1_4$ in figure 5.11), one for each of the existing operations in the "Display" class. Plan P1 posts event $c1'True$ which can be handled by three different plans P4, P5, and P6. However, plan type $P6$ does not generate any plan instance because its context condition does not hold (none of the operations *op* in the "Display" class has the same name as message *self*, i.e. "2:stream"). Therefore, there are 8 possible options to repair constraint $C1$("2 : *stream*") using plan type P1:

1. Rename operation "select()" to *stream*.
2. Rename operation "play()" to *stream*.
3. Rename operation "draw()" to *stream*.
4. Rename operation "stop()" to *stream*.

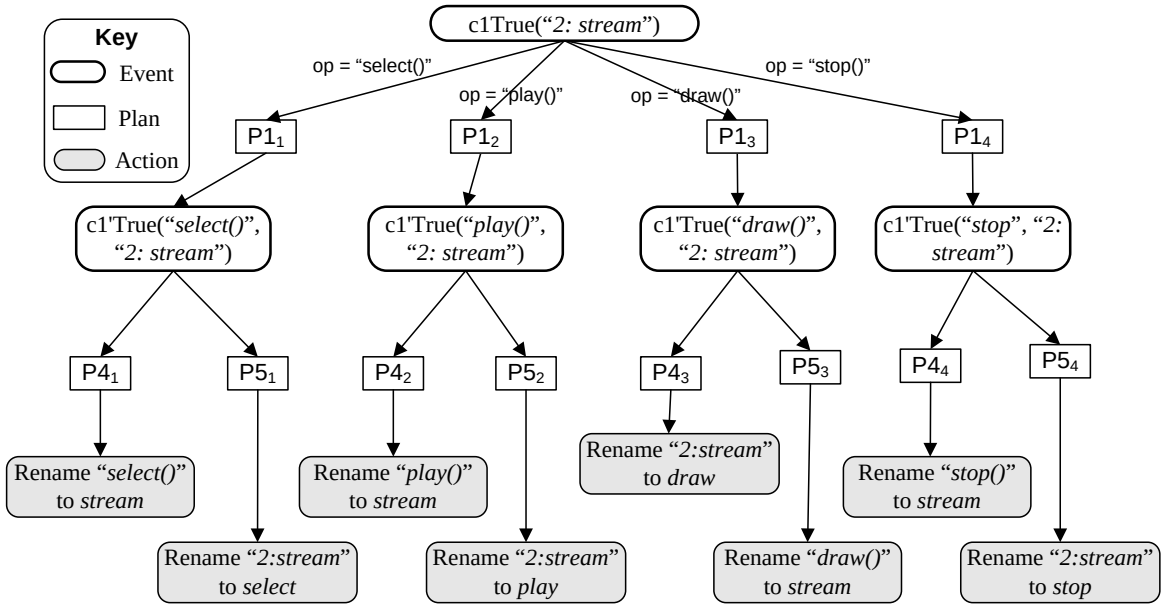


Figure 5.11: Repair plan instances for fixing constraint $C1(\text{“}2 : stream\text{”})$ *with respect to plan type* $P1$

5. Rename message “2:stream” to *select*.

6. Rename message “2:stream” to *play*.

7. Rename message “2:stream” to *draw*.

8. Rename message “2:stream” to *stop*.

Similarly, let us consider the instances generated by plan P2. Note that there are two existing operations that do not belong to the Display class: “stream()” and “wait()”. With regard to the case op = “*stream*()”, let us consider which plan instances are generated to handle the events posted within plan P2. Firstly, regarding the first event (*add op to self.receiver.base.operation*) plan P7 has one instance: connect operation “stream()” (of class “Streamer”) to class “Display”. Plan P8 has one instance (because x can only be class “Streamer”): posting an event *changing self.receiver.base to class “Streamer"*, which is handled by plans P9 and P10. Plan P9 has one instance: connect object “disp” to “Streamer” class and plan P10 has one instance: connect message “2:stream” with object “st:Streamer”. Finally, since the name of operation *op* (i.e. “stream”) is the same as the name of message *self*, plans P6 is applicable and which results in no further action being needed to handle

the second event ($c1'True(op, self)$) posted within plan P2.

Therefore, plan P2 with respect to op = "$stream()$" gives the following repair options to fix $C1$("$2 : stream$"):

9. Connect operation "stream()" to class "Display", i.e. add method "stream()" to class "Display".

10. Connect object "disp" to "Streamer" class, i.e. add class "Streamer" to the set of bases of object "disp".

11. Connect message "2:stream" with object "st:Streamer", i.e. changing the receiver of message "stream" to object "st".

Similarly, plan P2 with respect to op = "$wait()$" gives four repair options: two of them are described as below plus the other two are repair options 10 and 11 above (because these two options already make constraint $C1$("$2 : stream$") true and consequently plan P6 for $c1'True$("$wait()$", "$2 : stream$") is applicable which results in no further action being needed).

12. Connect operation "wait()" (of class "Streamer") to class Display, and rename operation "wait()" to *stream*.

13. Connect operation "wait()" (of class "Streamer") to class Display, and rename message "stream" to *wait*.

Plan P3 involves creation of a new operation and it has only one instance:

14. Create a new operation, add it to class "Display" and name it "stream".

Overall, there are 14 different options for fixing constraint $C1$("$2 : stream$"). Repair plan instances for the other three constraints are also created in a similar way.

Calculate cost[10]

The next process in our framework is calculating the cost of each repair option before presenting the cheapest ones to the user for selection. Firstly, in our framework the designer

[10]Here we briefly illustrate a mechanism for plan section based on cost calculation which applied for this example. A more detailed and formal discussion will be presented in chapter 7

Option	Cost
1	1 + C1(“1:select”) + C1(“3:play”) + C2(“disp”) + C2(“st”)
2	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
3	1 + C1(“3:play”) + C1(“4:draw”) + C2(“disp”) + C2(“st”)
4	1 + C1(“3:play”) + C1(“5:stop”) + C2(“disp”) + C2(“st”)
5	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
6	1 + C1(“3:play”) + C2(“st”)
7	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
8	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
9	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
10	1 + C1(“3:play”) + C2(“st”) + C2(“disp”)
11	1 + C1(“3:play”) + C2(“disp”) + C2(“st”)
12	2 + C1(“3:play”) + C1(“6:wait”) + C2(“disp”) + C2(“st”)
13	2 + C1(“3:play”) + C2(“disp”) + C2(“st”)
14	2 + C1(“3:play”) + C2(“disp”) + C2(“st”)

Table 5.1: The cost of repair options for fixing C1(“2:stream”)

is responsible for defining the costs for basic repair actions. The basic costs can be varied and may have some effects on which repair plans are considered as higher cost than others. For instance, in this example we use the following settings: the cost of creation is 0, the cost of connection, disconnection, and modification is 1, and the cost of deletion is 2. Other combinations can be used but may give a different outcome. Secondly, the cost of a repair option is the sum of the costs of its repair actions and the costs of fixing violated constraints existing after the repair option is executed. The former cost component is calculated by simply summing the cost of each primitive action in a repair option. On the other hand, in order to work out the latter cost we need to simulate the execution of a repair option.

For example, the cost of the first repair option (renaming operation “select()” to *stream*) is the cost of the renaming action (which is equal the cost of a modification, i.e. 1) plus the cost of fixing violated constraints. Assume that the first repair option is executed, then constraints C1(“3:play”), C2(“disp”) and C2(“st”) are still violated. In addition, there is one new violated constraints: C1(“1:select”) – the “1:select” does not match name with any operation in class “Display”. Therefore, the cost of the first repair option is 1 plus the costs of fixing constraints C1(“1:select”), C1(“3:play”), C4(“disp”), and C4(“st”). Table 5.1 shows the cost of the first repair option (the first row) as well as the cost of other repair options for fixing C1(“2:stream”).

The first repair option is an example where fixing an inconsistency may result in creating new inconsistencies. Repair option 6 is, on the other hand, an example where fixing an inconsistency may also lead to repairing other inconsistencies. In fact, this repair option fixes not only constraint C1("2:stream") but also constraint C2("disp").

In this example, it is easy to see that at this stage repair options 6 gives the cheapest cost. However, in other cases a full simulation, i.e. executing all repair plans, is needed to determine the cheapest repair option. Pruning techniques are also applied to improve the performance as well as to detect cycles, which are discussed in more detail in chapter 7. We briefly note here that repair option 6 (similarly to other repair options) needs to expand to include plans that fix constraint C1("3:play") and C2("st"). We then need to follow the same process to generate repair plan instances and calculate cost for those two constraints. However, it is easy to see that a cheapest repair option for both C1("3:play") and C2("st") is renaming message "3:play" to "stream".

Select one plan to execute and execute plan

After the cost of each repair plan is calculated, the next step is presenting the cheapest plans for user selection. However, since we have done full planning in the previous step, what we present to the user is repair plans that are able to fix not only one constraint but also all the constraints in the repair scope, e.g. C1("2:stream"), C1("3:play"), C2("disp"), and C2("st") in our example. The user chooses one of those repair plans and the framework will execute the plans to apply (secondary) changes to the model. These changes will make the model become consistent with respect to the repair scope. For instance, in our VOD example there is only one repair plan that has the cheapest cost of 2: renaming message "2:stream" to "play", and renaming message "3:play" to "stream". This repair plan is presented to the user and if it gets chosen it will be executed. The model will then become consistent with respect to constraints C1 and C2.

5.4 Chapter summary

In this chapter, we have investigated the applicability of our approach to UML using a small case study which concerns a small excerpt of the UML metamodel, a few consistency constraints and a simple application designed using UML. A similar setting has also been used in the recent work of Egyed [2007]. On the one hand, his work is similar to ours in

that he also aims to fix inconsistencies in UML design models. His approach uses model profiling to locate possible starting points for fixing an inconsistency in a UML model. He also tries to use model profiling to predict the side-effects of fixing an inconsistency. On the other hand, there are several major differences between his work and ours. Firstly, he treats a constraint as a black box whilst we analyse the constraints to generate repair plans. He does not provide options to repair inconsistencies, but only suggests starting points (entities in the model) for fixing the inconsistency. In other words, using his approach the designer would be advised where are the right places for resolving inconsistencies, but would not be told what the concrete fixes are. Our work goes further than that by suggesting the concrete feasible repair options.

With regard to this specific VOD example that is used in both his work and ours, the repair options that our framework identified match the choices of starting points suggested by his approach. The cheapest repair option from our framework's point of view, i.e. renaming message "2:stream" to "play" and renaming message "3:play" to "stream", is also the most reasonable option that the designer tends to follow. This example is an illustration of how our approach can be applied in the context of UML design models. Although the result seems promising, in order to fully understand the applicability of our approach to UML models, a more extensive investigation involving the whole UML metamodel and consistency constraints is needed. This is part of our future work. In addition, also as in [Egyed, 2007], the scalability issue of our approach should also be investigated, which will be discussed in chapter 9.

So far, we have presented a high level overview of our framework and used two case studies to illustrate how our ideas can be applied to different methodologies and design types. In the next two chapters, we will discuss the details of our framework in terms of how repair plans are automatically generated (chapter 6) and how to select repair plans for resolving a constraint violation (chapter 7).

Chapter 6

Plan Generation

There can be a substantial number of consistency constraints in the context of design models. For instance, we have identified nearly 50 consistency constraints that can be imposed on Prometheus design models. In UML 1.5 [Object Management Group, 2005], the number of well-formedness constraints is over 100. A consequence of large numbers of constraints is an even larger number of repair plans. In these cases, hand-crafting repair plans for all constraints becomes a labour intensive task. In addition, it is difficult for the repair administrator to know if the set of repair plans which they create is of high quality and not faulty, meaning that in practice they should be complete and correct. Therefore, we have developed a translation schema that takes constraints, expressed as OCL invariants, as input and generates repair plans that can be used to correct constraint violations. In this chapter we discuss the repair plan generation in detail. We first present a formal definition of repair actions. We then discuss some extensions to the abstract syntax of repair plans previously mentioned in chapter 3. In addition, we present the plan generation rules for a set of basic propositions in OCL as well as rules for all OCL boolean connectives (e.g. and, or, not, etc.). Using a combination of such rules, repair plans can be automatically generated for a range of OCL constraints. Finally, we explain and prove the correctness and completeness of our generated plans.

6.1 Formally defining repair actions

As previously discussed in section 3.1.1, we abstractly view a (design) model as consisting of a set of model entities and a set of relationships between entities. Model entities also have

attributes. We consider attributes that have primitive types (e.g. integer, string) instead of referencing types, which are regarded as relationships.

Each model entity is an instance of a metamodel element, which is also referred to as the entity's type. For instance, valid types in a Prometheus design model can be agent, plan, event, etc. A model entity is defined by a unique identifier (entity-id) and a type (entity-type). The set of entity-id and entity-type pairs in a design model is denoted as *entities*. For example, a model containing an agent and a plan has *entities* = {(e12, Agent), (e14, Plan)} where $e12$ and $e14$ are the unique identifiers of the agent and the plan respectively. An entity can only have a single type. A design model also contains a set of relationships between entities. It is noted that there can be more than one relationship between two entities. For example, between an agent and a message, in Prometheus, there can be three different types of relationships: an agent sends a message, an agent receives a message, and an agent owns a message. In addition, a relationship may involve more than two entities, i.e. n-ary relationships. However, n-ary relationships rarely exist compared to binary relationships and we can break an n-ary relationship into multiple binary relationships. Therefore, our focus is on binary relationships. We formally define a relationship as a triple: entity ID (source), relation ID, and entity ID (destination) and *relationships* is the set of those triples. For example, a model with an agent which owns a plan has *relationships* = {(e12, r1, e14)} where $r1$ is the unique identifier for the relationship between an agent and its belonging plan.

DesignModel

$$
\begin{aligned}
&\mathit{entities} : \mathit{IDENT} \pfun \mathit{TYPE} \\
&\mathit{relationships} : (\mathit{IDENT} \times \mathit{IDENT}) \times \mathit{IDENT} \\
&\mathit{attrValues} : \mathit{IDENT} \times \mathit{IDENT} \pfun \mathit{VALUE} \\
&\hline
&\mathrm{dom}(\mathrm{dom}\ \mathit{relationships}) \subseteq \mathrm{dom}\ \mathit{entities} \\
&\mathrm{ran}\ \mathit{relationships} \subseteq \mathrm{dom}\ \mathit{entities} \\
&\mathrm{dom}(\mathrm{dom}\ \mathit{attrValues}) \subseteq \mathrm{dom}\ \mathit{entites}
\end{aligned}
$$

We also formally represent the value of an entity's attribute as a value function *attrValues* from entity ID and attribute ID to a single value (e.g. integer, string), e.g. the *name* attribute of the entity *agent1* has the value "*Monitor Agent*". Overall, a design model is formally defined as a schema written in the Z notation [Spivey, 1989] as below[1]. Assume that *IDENT*

[1] In Z $\mathrm{dom}\ X$ is the domain of X and $A \oplus B = \{\langle x, y\rangle \mid \langle x, y\rangle \in A \wedge x \notin \mathrm{dom}\ B\} \cup B$

is the set of possible unique identifiers, $TYPE$ is the set of possible entity types, and $VALUE$ is the set of possible attribute values.

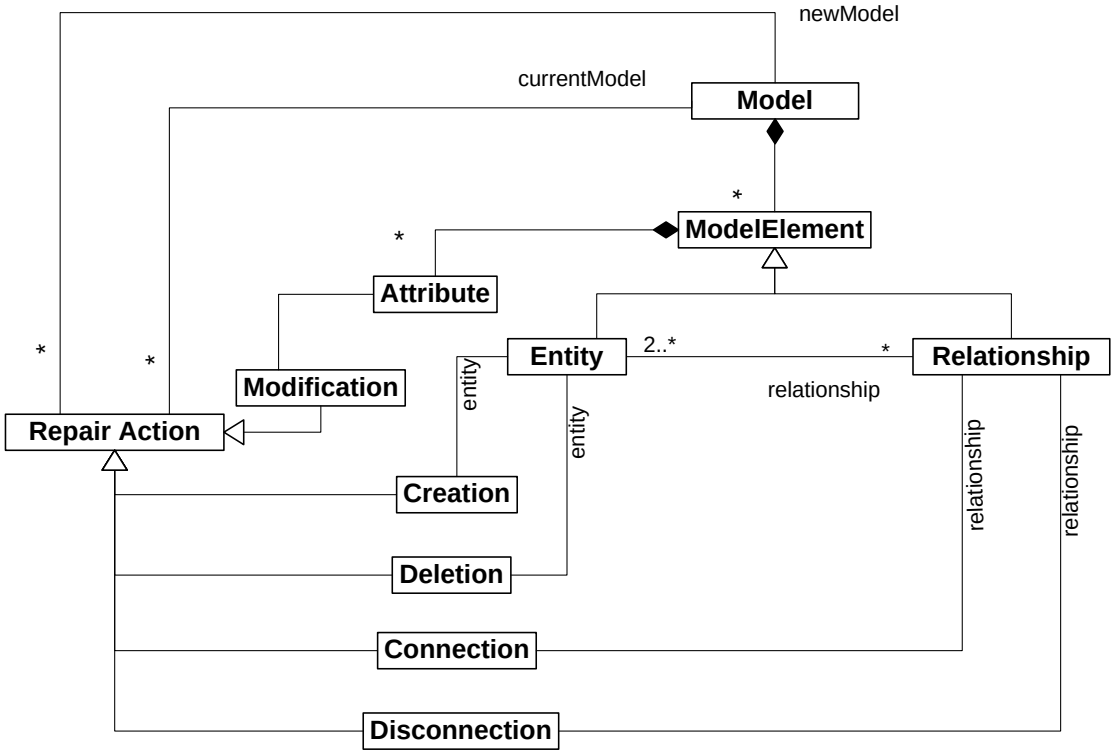


Figure 6.1: A taxonomy of repair actions

Previously in section 3.2.2 we have presented a classification of repair actions that we consider in our framework. The five types of primitive actions that are used to update the model are creation and deletion of entities, adding and removing relationships between entities (corresponding to connection and disconnection respectively), and modifying the values of attributes of entities (see figure 6.1). Each of them is formally defined below.

Create

$$\Delta DesignModel$$
$$eId? : IDENT$$
$$eType? : TYPE$$

$$eId? \notin \text{dom } entities$$
$$entities' = entities \cup \{(eId?, eType?)\}$$
$$relationships' = relationships$$
$$attrValues' = attrValues$$

A creation has two inputs: a unique identifier of a newly created entity (eId) and its type ($eType$). This action has a precondition that the new eId should be unique. The post condition is that the set of entities is updated with the new entity.

Deletion also requires two inputs: the entity to be deleted and its type. The precondition states that this entity to be deleted should exist. The postcondition is that the set of entities is updated excluding the entity to be removed. All attributes belonging to the entity to be removed are also deleted. In addition, the set of relationships is updated excluding any relationships involving the entity to be deleted[2].

Delete

$$\Delta DesignModel$$
$$eId? : IDENT$$
$$eType? : TYPE$$

$$(eId?, eType?) \in entities$$
$$entities' = entities \setminus \{(eId?, eType?)\}$$
$$relationships' = (\{eId?\} \lhd relationships) \rhd \{eId?\}$$
$$attrValues' = \{eId?\} \lhd attrValues$$

A connection has three inputs corresponding to two entities to be connected and a relationship between them. Its precondition requires that the two entities should exists. Its postcondition requires the set of relationships to be updated with the new relationship.

Connect

$$\Delta DesignModel$$
$$eId1?, eId2? : IDENT$$
$$rId? : IDENT$$

$$eId1? \in \mathrm{dom}\ entities$$
$$eId2? \in \mathrm{dom}\ entities$$
$$relationships' = relationships \cup \{(eId1?, rId?, eId2?)\}$$
$$entities' = entities$$
$$attrValues' = attrValues$$

A disconnection also has three inputs corresponding to two entities to be connected and a relationship between them. A disconnection is equivalent to removing a relationship from the

[2]In Z, $A \lhd B = \{\langle x, y\rangle \mid \langle x, y\rangle \in B \wedge x \notin A\}$ and $A \rhd B = \{\langle x, y\rangle \mid \langle x, y\rangle \in A \wedge y \notin B\}$

set of current relationships in a design model. Its precondition requires that the relationship should exist.

Disconnect

$\Delta DesignModel$
$eId1?, eId2? : IDENT$
$rId? : IDENT$

$relationships' = relationships \setminus \{(eId1?, rId?, eId2?)\}$
$entities' = entities$
$attrValues' = attrValues$

Modification involves an entity (eId), its attribute (aId) and a new value (val). It has a precondition requiring that the entity should exist and a postcondition requiring the entity's value be changed to val.

Modify

$\Delta DesignModel$
$eId? : IDENT$
$aId? : IDENT$
$val? : VALUE$

$eId? \in \text{dom } entities$
$attrValues' = attrValues \oplus \{(eId?, aId?) \mapsto val?\}$
$entities' = entities$
$relationships' = relationships$

6.2 Automatic repair plan generation: issues and solutions

The core part of the repair plan generator is a translation that takes an OCL constraint as input and generates repair plans that can be used to correct constraint violations. Such a translation can be developed by considering all the possible ways in which a constraint can be false, and hence all the possible ways in which it can be made true. However, there may be a large number of concrete ways of fixing a constraint violation. In some cases this number can be infinite. For instance, consider a constraint requiring a particular set SE to be non-empty. Assume that SE is empty, then the various ways of fixing this constraint are adding 1 element, adding 2 elements, adding 3 elements, and so on. Another example is a constraint requiring that the age (attribute) of a person (model entity) has to be greater than 18. There

is also an infinite number of ways to fix this constraint, each of which corresponds to changing the age to a number greater than 18. Such issues are due to the inherent characteristics of first order logic that OCL is based on. We avoid infinities by either ruling out the constraint forms that lead to an infinite number of repair options, or by consulting the user to provide further information to restrict the number of potential repair options.

We have addressed these issues when developing the repair plan translation in the following ways:

- Generated repair plans abstractly represent certain classes of concrete ways of fixing a constraint. For example, plan $c_t : x \in SE \leftarrow !c1_t(x)$ represents all the repair plans that make constraint c true, each of which corresponds to picking an element x in the set SE and making $c1(x)$ true.

- Generated repair plans are minimal in that all of their actions contribute towards fixing a certain constraint, i.e. taking out any of the actions results in failing to fix the constraint. For example, the generated repair plan for making a (empty) set be not empty is adding only one element to the set.

- There is a type of repair plan that involves user interaction in which the user is asked to provide an input. For example, in the above example the user is asked to input an age value that is greater than 18.

In the next section, we discuss several changes made to the abstract syntax of repair plans. Some of them are for accommodating the above solutions.

6.3 Extended repair plan syntax

The extended abstract syntax (see figure 6.2) has several modifications. Firstly, we also cover the case when the plan does nothing in which its plan body has only one construct called *true*. Secondly, we include the definition of repair actions as previously explained in section 6.1. "Create e : t" indicates a creation of entity e of type t whilst "Delete e" results in a deletion of entity e. "Connect $e1$ and $e2$ (w.r.t. r)" denotes a connection between entities $e1$ and $e2$ with respect to relationship r, and similarly "Disconnect $e1$ and $e2$ (w.r.t. r)" refers to a disconnection between entities $e1$ and $e2$ with respect to relationship r. "Change *attr* of e to *val*" involves modifying attribute *attr* of entity e to a new value *val*.

$$
\begin{array}{lll}
P & ::= & E[:\, C] \leftarrow B \\
E & ::= & C_t \mid C_f \mid +(x, SE) \mid -(x, SE) \\
C & ::= & C \vee C \mid C \wedge C \mid \neg\, C \mid \\
 & & \forall\, x \bullet C \mid \exists\, x \bullet C \mid \mathit{Prop} \mid \mathit{Ask}(\mathit{userVal},\ \mathit{guidance}) \\
B & ::= & \mathit{RepairAction} \mid !E \mid B_1;\ B_2 \mid \\
 & & \mathit{if}\ C\ \mathit{then}\ B \mid \mathit{for\ each}\ x\ \mathit{in}\ SE\ B \mid \mathit{true} \\
\mathit{RepairActions} & ::= & \mathit{Create}\ \mathrm{e} : \mathrm{t} \mid \mathit{Delete}\ \mathrm{e} \mid \mathit{Connect}\ \mathrm{e1}\ \mathit{and}\ \mathrm{e2}\ (w.r.t\ \mathrm{r}) \mid \\
 & & \mathit{Disconnect}\ \mathrm{e1}\ \mathit{and}\ \mathrm{e2}\ (w.r.t\ \mathrm{r}) \mid \mathit{Change}\ \mathrm{attr}\ \mathit{of}\ \mathrm{e}\ \mathit{to}\ \mathrm{val}
\end{array}
$$

Figure 6.2: Repair plan abstract syntax

Thirdly, the context condition can now contain a predicate $Ask(userVal, guidance)$ indicating that the user should be asked to provide (following the hints provided in the *guidance*) a value bounded to *userVal*. For example, the user is asked to provide a value for a new attribute. Finally, we define an event which can be: making a constraint true (C_t), or making a constraint false (C_f), or adding an entity to a derived set ($+(x, SE)$), or removing it from a derived set ($-(x, SE)$). Addition and deletion for derived sets (e.g. union sets) are represented as events because they do not usually involve a primitive action. Rather, they require a number of different actions, for example two deletions might be needed, one of the first set and the other for the second set, in the case of removing an entity from a union of two sets.

In the next sections, we will demonstrate how repair plans are written using this abstract syntax.

6.4 Plan generation rules

In this section, we present the rules which can be used to generate repair plans for a given consistency constraint expressed in OCL. Figure 6.3[3] depicts the grammar of the most important OCL expressions. Since the use of OCL in this case is to specify invariants on classes in the metamodel, our plan generation rules cover only OCL expressions in the form of invariants. As a result, OCL expressions for specifying constraints on operations, and pre- and post conditions on operations (i.e. *OperationContextDecl*) are not covered (although a similar approach can be adopted to cover those cases).

[3]Expressions and operators (e.g. *LetExp*, *AddOperator*) that we have not supported are in bold.

We cover OCL expressions that involve attributes (i.e. *AttributeCallExp*) and navigations (i.e. *NavigationCallExp*). In terms of navigations, the current rules support most of the navigations over associations (i.e. *AssociationEndCallExp*), except for navigation to/from association classes, and navigation through qualified associations (i.e. *AssociationClassCallExp*).

We support all the boolean connectives in OCL, i.e. *and*, *or*, *xor*, *not* and *implies* (i.e. *LogicalExp*). We do not cover the *let* expression which allows for defining an attribute or operation that can be used in a constraint (i.e. *LetExp*). However, such attributes or operations can be placed directly into a constraint. With respect to collection types, our repair plan generator covers only OCL *invariants* relating to all standard operations on *sets* including all predefined collection operations that use an iterator (i.e. *IteratorExp*), e.g. *select*, *collect*, *reject*, *forAll*, *exists*, etc. The two exceptions that we do not cover are the *iterate* (i.e. *IterateExp*) and *sum* operations. While the former is relatively complicated which requires future work, propositions derived from the latter (i.e. the *sum* operation) can have an infinite number of equivalent ways to repair and consequently user intervention would be very complicated. Hence, they can not be automatically generated. Although we do not have rules for other collection types, similar techniques can be used to derive repair plans for *bag* and *sequence* types.

In terms of basic values and types, the rules cover the four basic types pre-defined in OCL (Boolean, Integer, Real, and String). Although we do not explicitly address enumeration types as defined in OCL, we can consider it as a set of possible values, which can be expressed as a context condition of a repair plan. We do not address operations defined for OCL types such as *oclIsKindOf()*, *oclIsTypeOf()*, and *oclAsType()*, and leave them for future work.

There are OCL expressions that result in sets, e.g. navigation over associations with multiplicity more than one, or a *select* operation applied to a set returning another set, or other pre-defined set operations such as *union*, *intersection*, etc. (which are part of *OperationCallExp*[4]). Addition and deletion involving such derived sets are not simply primitive repair actions. In fact, we model them as events, which are handled by further repair plans. In summary, our plan generation rules address the following OCL expressions:

1. Navigation.

[4]An operation call can be a pre-defined collection operation (which we support), or other forms such as class operation call (which we do not cover).

ContextDeclaration	::=	*ClassifierContextDecl* | **OperationContextDecl**
ClassifierContextDecl	::=	"*context*" *pathName Invariant*
Invariant	::=	"*inv*" (*simpleName*)? : *OclExp Invariant*
OclExp	::=	**LetExp** | *LogicalExp*
LogicalExp	::=	*RelationalExp* (*LogicalOperator RelationalExp*)∗
RelationalExp	::=	*AdditiveExp* (*RelationalOperator AdditiveExp*)?
AdditiveExp	::=	*MultiplicativeExp* (*AddOperator MultiplicativeExp*)∗
MultiplicativeExp	::=	*UnaryExp*(*MultiplyOperator UnaryExp*)∗
UnaryExp	::=	(*UnaryOperator PostfixExp*) | *PostfixExp*
PostfixExp	::=	*PrimaryExp* | ((“.” | “− > ”) *ModelPropertyCallExp*)∗
PrimaryExp	::=	**LiteralExp** | *ModelPropertyCallExp* | **IfExp**
ModelPropertyCallExp	::=	*OperationCallExp* | *AttributeCallExp*
		NavigationCallExp | *LoopExp*
NavigationCallExp	::=	*AssociationEndCallExp* | *AssociationClassCallExp*
LoopExp	::=	*IteratorExp* | **IterateExp**
OperationCallExp	::=	*OclExp simpleName OclExp* |
		OclExp “ → ” *simpleName* “(” *arguments*? “)”
		OclExp “.” *simpleName* “(” *arguments*? “)”
		simpleName “(” *arguments*? “)”
		pathName “(” *arguments*? “)”
		simpleName OclExp
AttributeCallExp	::=	*OclExp* “.” *simpleName* | *simpleName* | *pathName*
IfExp	::=	“*if*” *OclExp* “*then*” *OclExp*
		“*else*” *OclExp* “*endif*”
IteratorExp	::=	*OclExp* “ → ” *Iterator* “(” *VariablesDecl* “ | ” *OclExp* “)”
IterateExp	::=	*OclExp* “ → ” “*iterate*” “(” *VariablesDecl* “ | ” *OclExp* “)”
Iterator	::=	“*forAll*” | “*exists*” | “*select*” | “*reject*” | “*one*” |
		“*collect*” | “*any*” | “*isUnique*” | “**sortedBy**”
LiteralExp	::=	**EnumLiteral** | *CollectionLiteral* | *PrimitiveLiteral*
LogicalOperator	::=	“*and*” | “*or*” | “*xor*” | “*implies*”
RelationalOperator	::=	“ = ” | “ <> ” | “ < ” | “ > ” | “ <= ” | “ >= ”
UnaryOperator	::=	“ − ” | “*not*”
AddOperator	::=	“ + ” | “ − ”
MultiplyOperator	::=	“ × ” | “ : ”

Figure 6.3: An excerpt of the OCL grammar (adopted from [Object Management Group, 2006] and [Object Management Group, 2005])

2. Constraints on attributes (e.g. <, >, <>, and =).

3. Constraints on Boolean-valued set expressions (e.g. *forAll*, *exists*).

4. Constraints on non-Boolean-valued set expressions.

5. Boolean connectives (e.g. *and*, *or*, etc.).

6. Addition or deletion involving derived sets.

It is noted that all of the consistency constraints that we have identified in Prometheus can be expressed using the above forms of OCL expressions that the plan generation rules address. With respect to the UML, most of the consistency constraints are (or can be, refer to [Object Management Group, 2005]) defined using those forms. However, we also consider to fully support all forms of OCL expressions, which is part of our future work. We now explain and discuss the rules that we currently support in detail.

Firstly, it it noted that consistency constraints describe properties that must be true about the application model. Such a constraint, however, can contain many individual clauses (i.e. sub-constraints) that can be true or false, e.g. logical connectives such as *not*, *xor*, etc. Hence, we need to consider repair plans of making each possible clause true or false, which can then be combined to form repair plans for making the top-level constraint true. For example, repairing constraint $c_1 \stackrel{\text{def}}{=} not\ c_2$ requires us to look for assignments that make sub-constraint c_2 false. Therefore, our plan generation has rules that make a constraint true c_t (under the assumption that the constraint has been violated) and rules that make a constraint false c_f (under the assumption that the constraint will evaluate to true). We now explain the plan generation rules, for both c_t and c_f, for each of the supported basic OCL propositions.

The translation for this group of rules is defined as a function $\mathcal{P}$ that takes an OCL constraint as input and returns a set of specific plans that are based on the constraint form. We now define function $\mathcal{P}$ in terms of plan generation rules for each of the basic OCL propositions that are covered.

6.4.1 Navigation

A common expression in OCL is navigations over an association, which applies to an entity and results in either another single entity or a set of entities, depending on the nature of

the association. In the case where the multiplicity of an association end is one (or zero), the result of its corresponding navigation is either a single entity or null. In the case where the multiplicity of the association end is more than one, the result of its corresponding navigation is a set of entities (which we also consider as derived sets). In this section we present rules that cover navigations leading to a single entity. In this context, equality is the only valid relation operator. In section 6.4.6, we have rules that cover the case of navigations leading to multiple entities.

Assume that E and x are two given model entities, and $E.aend$ navigates the association by using the opposite association end[5] $aend$, leading to zero or one entities. We identify the following basic propositions:

- $\mathbf{c \stackrel{def}{=} E.aend = x}$: This constraint returns true if entities E and x are connected with respect to association end $aend$. If E is currently not connected to any entity with respect to association end $aend$ (i.e. E.aend = null), then to make this constraint true we connect E and x with respect to association end $aend$. Otherwise, assume that E is connected to y with respect to association end $aend$ (i.e. E.aend = y[6]), then we have to disconnect E from y before connecting E and x. To make this constraint false, we just disconnect E from x.

- $\mathbf{c \stackrel{def}{=} E.aend1.aend2 = x}$: This constraint returns true if entities $E.aend1$ (if $E.aend1$ refers to a single entity[7]) and x are connected with respect to association end $aend2$. To make this constraint true, we need to ensure that $E.aend1 = z$ for some z and that $z.aend2 = x$. There are several specific possible patterns of action sequences depending on the choice of z: z can be a newly created entity, or it can be an existing entity. In the latter case there are two distinct situations: z can be the (unique) entity for which $E.aend1 = z$ currently holds, or it can be another existing entity (which can be either connected to x w.r.t. $aend2$, i.e. $z.aend2 = x$, or not connected to x w.r.t. $aend2$ but having the same type of $E.aend1$, i.e. $z \in Type(E.aend1)$). To make this constraint false, we either disconnect $E.aend1$ from x or disconnect E from y where $y.aend2 = x$.

[5] In OCL, a navigation expression may start with an object and then navigate to a connected object by referencing the latter by the role name attached to its association end. More details and examples of OCL syntax have been discussed in section 2.3 on page 39.

[6] In this case, $E.aend$ is implicity not $null$, i.e. $y \neq null$.

[7] Note that $E.aend1$ is not null, otherwise $E.aend1.aend2$ is not valid since "any property call applied on null results in OclInvalid, except for the operation oclIsUndefined()" [Object Management Group, 2006].

- **c** $\stackrel{\text{def}}{=}$ **E1.aend1 = E2.aend2**: This constraint returns true if entities $E1$ and $E2$ connect to the same entity with respect to association ends $aend1$ and $aend2$ respectively. To make this constraint true, we need to ensure that $E1.aend1 = E2.aend2$ which necessarily involves ensuring either that $E1.aend1 = x$ and $E2.aend2 = x$ for some x. There are four specific possible patterns of action sequences depending on the choice of x : x can be a newly created entity, or it can be an existing entity[8]. In the latter case there are three distinct situations: x can be the (unique) entity for which either $E1.aend1 = x$ or $E2.aend2 = x$ currently holds, or it can be another existing entity. To make this constraint false, we change either $E1.aend1$ or $E2.aend2$ so that they are different from each other.

$\mathcal{P}_t$(**c** $\stackrel{\text{def}}{=}$ **E.aend = x**) =
  {c_t : E.aend = null $\leftarrow$ Connect E and x (w.r.t *aend*),
  c_t : E.aend = y $\leftarrow$ Disconnect E and y (w.r.t *aend*); Connect E and x (w.r.t. *aend*)}
$\mathcal{P}_t$(**c** $\stackrel{\text{def}}{=}$ **E.aend1.aend2 = x**) =
  {c_t : E.aend1 = y $\leftarrow$!$(y.aend2 = x)_t$,
  c_t : z.aend2 = x $\leftarrow$!$(E.aend1 = z)_t$,
  c_t : z $\in$ Type(E.aend1) $\wedge$ z.aend2 $\neq$ x $\leftarrow$!$(E.aend1 = z)_t$;!$(z.aend2 = x)_t$,
  $c_t \leftarrow$ Create z : Type(E.aend1); !$(E.aend1 = z)_t$; !$(z.aend2 = x)_t$ }
$\mathcal{P}_t$(**c** $\stackrel{\text{def}}{=}$ **E1.aend1 = E2.aend2**) =
  {c_t : E2.aend2 = x $\leftarrow$!$(E1.aend1 = x)_t$,
  c_t : E1.aend1 = x $\leftarrow$!$(E2.aend2 = x)_t$,
  c_t : x $\in$ Type(E1.aend1) $\wedge$ x $\neq$ E1.aend1 $\wedge$ x $\neq$ E2.aend2
    $\leftarrow$!$(E1.aend1 = x)_t$; !$(E2.aend2 = x)_t$,
  $c_t \leftarrow$ Create x : Type(E1.aend1); !$(E1.aend1 = x)_t$; !$(E2.aend2 = x)_t$}

Figure 6.4: Plan generation rules (c_t) for basic propositions involving navigations to a single entity

Figures 6.4 and 6.5 give the definition of functions $\mathcal{P}_t$ and $\mathcal{P}_f$ for the above three propositions corresponding to making the input constraint true or false respectively. We can apply those rules for any multiple navigations that lead to a single entity. For instance, regarding a constraint of the form of $E1.aend1.aend2 = E2.aend3$, we can either change $E1.aend1.aend2$ to match the value of $E2.aend3$ (i.e. repair plans are generated by applying the rule $E1.aend1.aend2 = x$) or vice versa (repair plans are generated by applying the rule

[8]We assume that a solution that sets both $E1.aend1$ and $E2.aend2$ to *null* is not desirable, but that if it is acceptable, then it can be added as an extra repair plan.

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend = x}) =$
$\{c_f \leftarrow$ Disconnect E and x (w.r.t $aend)\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend1.aend2 = x}) =$
$\{c_f \leftarrow$ Disconnect E.aend1 and x (w.r.t $aend2)$,
$c_f \leftarrow$ Disconnect E and E.aend1 (w.r.t. $aend1)\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.aend1 = E2.aend2}) =$
$\{c_f :$ E2.aend = x $\leftarrow (E1.aend = x)_f$,
$c_f :$ E1.aend = x $\leftarrow (E2.aend = x)_f\}$

Figure 6.5: Plan generation rules (c_f) for basic propositions involving navigation to a single entity

$E2.aend3 = x$) or changing both $E1.aend1.aend2$ and $E2.aend3$ to another same entity.

6.4.2 Constraints on attributes

OCL uses $E.a$ to denote the attribute a of entity E. The value of $E.a$ can be of four basic types supported in OCL: *Boolean*, *Integer*, *Real* and *String*. For attribute values, we consider the following basic propositions. Note that we do not have a rule for inequality proposition since we consider it as a case of a *not* of an equality constraint.

- $\mathbf{c} \stackrel{\text{def}}{=} E.attr = val$: is true if the attribute *attr* of entity E has a value of *val*. To make this constraint true we simply change the value of *attr* to *val*. To make it false, we change it to any value that is different to *val*. Since there is a large number of such values (except if *val* is a Boolean), the user must be asked to provide a value.

- $\mathbf{c} \stackrel{\text{def}}{=} E.attr > val$: is true if the attribute *attr* of entity E has a value greater than *val*. To make this constraint true we need to change the value of *attr* to a value provided by the user that is greater than *val*. To make this constraint false we need to change the value of *attr* to a value provided by the user that is smaller or equal to *val*.

- $\mathbf{c} \stackrel{\text{def}}{=} E.attr < val$: is true if the attribute *attr* of entity E has a value smaller than *val*. To make this constraint true we need to change the value of *attr* to a value provided by the user that is smaller than *val*. To make this constraint false we need to change the value of *attr* to a value provided by the user that is greater or equal to *val*.

- $\mathbf{c} \stackrel{\text{def}}{=} E1.attr1 = E2.attr2$: is true if the attribute $attr1$ of entity $E1$ has the same value as the attribute $attr2$ of entity $E2$. To make this constraint true we need to ensure that

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr = val}) =$
$\{c_t \leftarrow$ Change $attr$ of E to $val\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr > val}) =$
$\{c_t :$ Ask(userVal, "> val") $\wedge$ userVal $>$ val $\leftarrow$ Change $attr$ of E to $userVal\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr < val}) =$
$\{c_t :$ Ask(userVal, "< val") $\wedge$ userVal $<$ val $\leftarrow$ Change $attr$ of E to $userVal\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.attr1 = E2.attr2}) =$
$\{c_t :$ E1.attr1 = val $\leftarrow !(E2.attr2 = val)_t$,
$c_t :$ E2.attr2 = val $\leftarrow !(E1.attr1 = val)_t$,
$c_t :$ Ask(userVal, "$\neq$ E1.attr1 and $\neq$ E2.attr2") $\wedge$ userVal $\neq$ E1.attr1 $\wedge$ userVal $\neq$ E2.attr2
$\leftarrow !(E1.attr1 = userVal)_t$; $!(E2.attr2 = userVal)_t\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.attr1 > E2.attr2}) =$
$\{c_t :$ E1.attr1 = val $\leftarrow !(E2.attr < val)_t$,
$c_t :$ E2.attr2 = val $\leftarrow !(E1.attr1 > val)_t\}$

Figure 6.6: Plan generation rules (c_t) for basic propositions involving attributes

$E1.attr1 = E2.attr2$ which necessarily involves ensuring either that $E1.attr1 = val$ and $E2.attr2 = val$. There are three specific possible patterns of action sequences depending on the choice of val: it is either the current value of $E1.attr1$ or the current value of $E2.attr2$ or a new value provided by the user (i.e. $userVal$. To make this constraint false we either change the value of $attr2$ to a value that is different of $attr1$ or vice versa.

- $\mathbf{c} \stackrel{\text{def}}{=} E1.attr1 > E2.attr2$: Returns true if the value of attribute $attr1$ of entity $E1$ is greater than the value of attribute $attr2$ of entity $E2$. To make this constraint true we either change the value of $attr1$ to a value greater than $attr2$ or change the value of $attr2$ to a value smaller than $attr1$. To make this constraint false we either change the value of $attr1$ to a value less than or equal to the value of $attr2$ or change the value of $attr2$ to a value greater of equal to the value of $attr1$.

Figures 6.6 and 6.7 give the definition of functions $\mathcal{P}_t$ and $\mathcal{P}_f$ for the above propositions corresponding to making the input constraint true or false respectively. Note that there is a predicate $Ask(userVal, guidance)$ in the context condition of some repair plans indicating that the user is asked to provide a value for $userVal$. The predicate also indicates some hints in terms of how the expected value should satisfy a certain condition provided in the $guidance$. For instance, $Ask(userVal, \text{“} > val\text{”})$ implies that the user should input a value

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr = val}) =$
$\{c_f$: Ask(userVal, "$\neq$ val") $\wedge$ userVal $\neq$ val$\leftarrow$ Change *attr* of E to *userVal*$\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr > val}) =$
$\{c_f$: Ask(userVal, "$\leq$ val") $\wedge$ userVal $\leq$ val $\leftarrow$ Change *attr* of E to *userVal*$\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr < val}) =$
$\{c_f$: Ask(userVal, "$\geq$ val") $\wedge$ userVal $\geq$ val $\leftarrow$ Change *attr* of E to *userVal*$\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.attr1 = E2.attr2}) =$
$\{c_f$: E1.attr1 = val $\leftarrow$ $!(E2.attr2 = val)_f$,
c_f : E2.attr2 = val $\leftarrow$ $!(E1.attr1 = val)_f\}$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.attr1 > E2.attr2}) =$
$\{c_f$: E1.attr1 = val $\leftarrow$ $!(E2.attr < val)_f$,
c_f : E2.attr2 = val $\leftarrow$ $!(E1.attr1 > val)_f\}$

Figure 6.7: Plan generation rules (c_f) for basic propositions involving attributes

that is greater than *val*. The guidance string is also checked by the generated plan.

6.4.3 Constraints on Boolean-valued set expressions

We cover all set operations returning boolean values supported in OCL (except *isUnique()*).

- $\mathbf{c} \stackrel{\text{def}}{=}$ **SE→includes(x)**: Returns true if x is an element of set SE, false otherwise. To make the constraint true, we simply add x to SE (which may be able to be done in a number of ways depending on SE), and to make it false we remove x from SE.

- $\mathbf{c} \stackrel{\text{def}}{=}$ **SE→excludes(x)**: Returns true if x is not an element of SE, false otherwise. To make this constraint true, we simply remove x from SE, and to make it false we add x to set SE.

- $\mathbf{c} \stackrel{\text{def}}{=}$ **SE→includesAll(SE')**: Returns true if the set SE contains all of the elements of the set SE'. To make this constraint true we pick out all the elements that are present in SE' but not in SE and for each such element either remove it from SE' or add it to SE. To make the constraint false we either pick an element in SE' and remove it from SE[9], or add an existing entity not belonging to SE to SE', or create a new entity and add it to SE'.

[9]Since the constraint is true, i.e. $SE' \subseteq SE$, this element is in SE.

- **c** $\stackrel{\text{def}}{=}$ **SE→excludesAll(SE')**: Returns true if SE contains none of the elements of SE' (this constraint is equivalent to $SE \cap SE' = \varnothing$). To make this constraint true we, for each element that is in both SE and SE', remove it from either SE or SE'. There are several ways of making the constraint false: adding an element in SE' (but not in SE) to SE, adding an element in SE (but not in SE') to SE', adding an existing element (not in SE and SE') to both SE and SE', or creating a new element and adding it to both SE and SE'.

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)}) =$ $\{c_t \leftarrow !{+}(\text{x, SE})\}$	$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}notEmpty()}) =$ $\{c_t : \text{x} \in \text{Type(SE)} \leftarrow !{+}(\text{x, SE}),$ $c_t \leftarrow \text{Create x : Type(SE) ; } !{+}(\text{x, SE})\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includesAll(SE')}) =$ $\{c_t \leftarrow \text{for each x in (SE' − SE) } !c'_t(\text{x}),$ $c'_t(\text{x}) \leftarrow !{-}(\text{x, SE'}),$ $c'_t(\text{x}) \leftarrow !{+}(\text{x, SE})\}$	$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}forAll(c1)}) =$ $\{c_t \leftarrow \text{for each x in SE if } \neg \text{ c1(x) then } !c'_t(\text{x}),$ $c'_t(\text{x}) \leftarrow !{-}(\text{x, SE}),$ $c'_t(\text{x}) \leftarrow !c1_t(\text{x})\}$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)}) =$ $\{c_t \leftarrow !{-}(\text{x, SE})\}$	$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}exists(c1)}) =$ $\{c_t : \text{x} \in \text{SE} \leftarrow !c1_t(\text{x}),$ $c_t : \text{x} \in \text{Type(SE)} \wedge \text{x} \notin \text{SE} \leftarrow !{+}(\text{x, SE}) ; !c1_t(\text{x}),$ $c_t \leftarrow \text{Create x : Type(SE) ; } !{+}(\text{x, SE}) ; !c1_t(\text{x}),$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludesAll(SE')}) =$ $\{c_t \leftarrow \text{for each x in SE} \cap \text{SE' } !c'_t(\text{x})$ $c'_t(\text{x}) \leftarrow !{-}(\text{x, SE'}),$ $c'_t(\text{x}) \leftarrow !{-}(\text{x, SE})\}$	$c1_t(\text{x}) : \text{c1(x)} \vee \text{c} \leftarrow \text{true } \}$ $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE1 = SE2}) = \mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty()})$ where SE = (SE1 − SE2)→union(SE2 − SE1)
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty()}) =$ $\{c_t \leftarrow \text{for each x in SE } !{-}(\text{x, SE})\}$	$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}one(c1)}) = \mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE'{\rightarrow}size() = 1})$ where SE' = SE→select(c1)

Figure 6.8: Plan generation rules (c_t) for basic propositions involving set operations returning boolean values

- **c** $\stackrel{\text{def}}{=}$ **SE→isEmpty()**: Returns true if SE contains no elements. To make the constraint true we remove all the elements in SE. To make it false we either add an existing element to SE or create a new element and then add it to SE.

- **c** $\stackrel{\text{def}}{=}$ **SE→notEmpty()**: Returns true if SE contains one or more elements. To make the constraint true we either add an existing element to SE or create a new element and then add it to SE. To make this constraint false, we remove all the elements in

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)}) =$
$\{c_f \leftarrow$!-(x, SE)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includesAll(SE')}) =$
$\{c_f$: x $\in$ SE' $\leftarrow$!-(x, SE),
c_f : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE'),
$c_f \leftarrow$ Create x : Type(SE) ; !+(x, SE')$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)}) =$
$\{c_f \leftarrow$!+(x, SE)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludesAll(SE')}) =$
$\{c_f$: x $\in$ SE' $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE),
c_f : x $\in$ SE $\wedge$ x $\notin$ SE' $\leftarrow$!+(x, SE'),
c_f : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\wedge$ x $\notin$ SE' $\leftarrow$
!+(x, SE) ; !+(x, SE'),
$c_f \leftarrow$ Create x : Type(SE) ;
!+(x, SE); !+(x, SE')$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}notEmpty()}) =$
$\{c_f \leftarrow$ for each x in SE !-(x, SE)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty()}) =$
$\{c_f$: x $\in$ Type(SE) $\leftarrow$!+(x, SE),
$c_f \leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}forall(c1)}) =$
$\{c_f$: x $\in$ SE $\leftarrow$!$c1_f$(x),
c_f : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE) ; !$c1_f$(x),
$c_f \leftarrow$ Create x : Type(SE) ; !+(x, SE) ; !$c1_f$(x),
$c1_f$(x) : $\neg$ c1(x) $\vee \neg$ c $\leftarrow$ true $\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}exist(c1)}) =$
$\{c_f \leftarrow$ for each x in SE if c1(x) then !c'_f(x),
c'_f(x) $\leftarrow$!-(x, SE),
c'_f(x) $\leftarrow$!$c1_f$(x)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE1 = SE2}) = \mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty()})$
where SE = (SE1 $-$ SE2)$\rightarrow$union(SE2 $-$ SE1)

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}one(c1)}) = \mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE'{\rightarrow}size() = 1})$
where SE' = SE$\rightarrow$select(c1)

Figure 6.9: Plan generation rules (c_f) for basic propositions involving set operations returning boolean values

SE.

- **c** $\stackrel{\text{def}}{=}$ **SE→exist(c1)**: Returns true if there is at least one element in *SE* for which constraint $c1$ is true. There are three ways of making this constraint true: pick one element in *SE* and make constraint $c1$ true for this element, add an existing element (not in *SE*) to *SE* and make the constraint true for it, and create a new element, add it to *SE*, and make the constraint true for it. To make this constraint false, we pick out all the elements in *SE* that make $c1$ true, and for each such element either delete it from *SE* or make $c1$ false for it.

- **c** $\stackrel{\text{def}}{=}$ **SE→forAll(c1)**: Returns true if constraint $c1$ is true for all elements in *SE*. To make this constraint true, we choose all the elements in *SE* that make $c1$ false, and for each such element either delete it from *SE* or make $c1$ true for it. To make this constraint false, we pick out an element in *SE* and make $c1$ false for it, or add an

existing element to SE and make $c1$ for it, or create a new element, add it to SE and make $c1$ false for it.

- **c** $\stackrel{\text{def}}{=}$ **SE→one(c1)**: Returns true if there is exactly one element in SE for which the constraint $c1$ is true. This constraint is equivalent to SE→select(c1)→size() = 1. We can consider SE→select(c1) as a (derived) set (refer to section 6.4.6) and we also have a rule for generating constraints in the form of SE→size() = m that is discussed in section 6.4.4.

- **c** $\stackrel{\text{def}}{=}$ **SE1 = SE2**: Returns true if $SE1$ and $SE2$ contain the same elements. This constraint is equivalent to SE→isEmpty() where SE = (SE1−SE2)→union(SE2−SE1). This constraint form is addressed in section 6.4.6 on page 149.

Figures 6.8 and 6.9 gives generation rules for the above cases. The terms SE, SE', $SE1$, and $SE2$ denote a set expression which can be a single set (e.g. the set of agents in the system) or a derived set which is built from another set using a navigation (e.g. S $\stackrel{\text{def}}{=}$ *self.agent*) or set operation (e.g. S $\stackrel{\text{def}}{=}$ S1→select(c)). Adding an element and removing an element from a derived set are considered as events $+(x, SE)$ and $-(x, SE)$ respectively. We will discuss how plans are generated to handle such events in section 6.4.6.

6.4.4 Constraints on non-Boolean-valued set expressions

In addition to operations that return boolean values, there are several set operations that result in other primitive values, which form the following basic propositions.

- **SE→size()**: The number of elements in SE. Constraints can be derived: *SE→size() = m*, *SE→size() > m*, and *SE→size() < m*. For a constraint in the form of *SE→size() = SE'→size()* (similarly to inequality constraints, e.g. *SE→size() > SE'→size()*), we can generate repair plans by changing either SE so that *SE→size()* is m (where m = *SE'→size()*) or vice versa or changing both SE and SE' in such a way that they have the same size.

- **SE→count(x)**: The number of times that element x occurs in SE. Since SE is a set, the value of the operation is at most 1. Therefore, two constraints can be derived including *SE→count(x) = 1* and *SE→count(x) = 0*, which are equivalent to *SE→includes(x)* and *SE→excludes(x)* respectively.

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() = m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$|m - s|$}$\}$ $!c'_t$,
c'_t : m < s $\wedge$ x $\in$ SE $\leftarrow$!$-$(x, SE),
c'_t : m > s $\wedge$ x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE),
c'_t : m > s $\leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$
where s = SE$\rightarrow$size()

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() \geq m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$(m - s)$}$\}$ $!c'_t$,
c'_t : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE),
$c'_t \leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$
where s = SE$\rightarrow$size()

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() \leq m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$(s - m)$}$\}$
$!c'_t$,
c'_t : x $\in$ SE $\leftarrow$!$-$(x, SE)$\}$
where s = SE$\rightarrow$size()

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}count(x) = 0}) =$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)})$

$\mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}count(x) = 1}) =$
$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)})$

Figure 6.10: Plan generation rules (c_t) for basic propositions involving set operations returning primitive values

We have generation rules for propositions related to two operations *size*() and *count*() (see figures 6.10 and 6.11).

6.4.5 Boolean connectives

We support all the boolean connectives in OCL, i.e. *and*, *or*, *xor*, *not* and *implies*.

- **c $\stackrel{\text{def}}{=}$ c1 and c2**: Returns true if both $c1$ and $c2$ are true. To make this constraint true, we need to make either $c1$ or $c2$ true , depending on which one (or both) is false. If $c1$ and $c2$ are both false initially, we can fix one of them first (without loss of generality assuming $c1$) and then fix the other one (i.e. $c2$) if it has not been fixed yet. In section 6.4.7, we introduce some top level plans (i.e. $fixC_t$) to deal with these cases. To make the constraint false, we make one of the sub-constraints false.

- **c $\stackrel{\text{def}}{=}$ c1 or c2**: Returns true if either $c1$ or $c2$ is true. To make this constraint true we make either constraint $c1$ or $c2$ true. To make the constraint false, we need to make either $c1$ or $c2$ false, depending on which one is true, or make both of them false if they are both true.

- **c $\stackrel{\text{def}}{=}$ c1 xor c2**: Returns true if either $c1$ or c is true but not both. To make this constraint true, we either make one of them true if they are both false or make one of

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() = m}) =$
$\{c_f$: x ∈ SE ← !−(x, SE),
c_f : x ∈ Type(SE) ∧ x ∉ SE ← !+(x, SE),
c_f ← Create x : Type(SE) ; !+(x, SE)$\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() < m}) =$
$\{c_f \leftarrow$ for each i in {1,2,...,$(m - s)$}$\}$ $!c'_f$,
c'_f : x ∈ Type(SE) ∧ x ∉ SE ← !+(x, SE),
c'_f ← Create x : Type(SE) ; !+(x, SE)$\}$
where s = SE→size()

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() > m}) =$
$\{c_f \leftarrow$ for each i in {1,2,...,$(s - m)$}$\}$
$!c'_f$,
c'_f : x ∈ SE ← !−(x, SE)$\}$
where s = SE→size()

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}count(x) = 0}) =$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)})$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}count(x) = 1}) =$
$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)})$

Figure 6.11: Plan generation rules (c_f) for basic propositions involving set operations returning primitive values

them false if they are both true. To make this constraint false, we either make the true constraint false or make the false constraint true.

- **c** $\stackrel{\text{def}}{=}$ **not c1**: Returns true if constraint $c1$ is false. This constraint is made true by making $c1$ false and is made false by making $c1$ true.
- **c** $\stackrel{\text{def}}{=}$ **c1 implies c2**: Returns true if either $c1$ is false or $c2$ is true. To make this constraint true we either make $c1$ false or make $c2$ true. To make the constraint false we make $c2$ false if $c1$ is true, or make $c1$ true if both of them are false, or make $c1$

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ and\ c2}) =$
$\{c_t$: ¬ c1 ∧ c2 ← $!c1_t$,
c_t : ¬ c2 ∧ c1 ← $!c2_t$,
c_t : ¬ c1 ∧ ¬ c2 ← $!c1_t$ $\}$

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ or\ c2}) =$
$\{c_t \leftarrow !c1_t$,
$c_t \leftarrow !c2_t\}$

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ xor\ c2}) =$
$\{c_t$: c1 ∧ c2 ← $!c1_f$,
c_t : c1 ∧ c2 ← $!c2_f$,
c_t : ¬ c1 ∧ ¬ c2 ← $!c1_t$,
c_t : ¬ c1 ∧ ¬ c2 ← $!c2_t\}$

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{not\ c1}) = \{\ c_t \leftarrow !c1_f\}$

$\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ implies\ c2}) = \{\ c_t \leftarrow !c1_f$,
$c_t \leftarrow !c2_t\}$

Figure 6.12: Plan generation rules (c_t) for boolean connectives

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ and\ c2}) =$
$\{c_f \leftarrow !c1_f,$
$c_f \leftarrow !c2_f\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ or\ c2}) =$
$\{c_f : \neg\, \text{c1} \wedge \text{c2} \leftarrow !c2_f,$
$c_f : \neg\, \text{c2} \wedge \text{c1} \leftarrow !c1_f,$
$c_f : \text{c1} \wedge \text{c2} \leftarrow !c1_f\ ;\ !c2_f\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{not\ c1}) = \{c_f \leftarrow !c1_t\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ xor\ c2}) =$
$\{c_f : \text{c1} \wedge \neg\, \text{c2} \leftarrow !\ c1_f,$
$c_f : \text{c1} \wedge \neg\, \text{c2} \leftarrow !c2_t,$
$c_f : \neg\, \text{c1} \wedge \text{c2} \leftarrow !c1_t,$
$c_f : \neg\, \text{c1} \wedge \text{c2} \leftarrow !c2_f\}$

$\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ implies\ c2}) =$
$\{\ c_f : \text{c1} \wedge \text{c2} \leftarrow !\ c2_f,$
$c_f : \neg\, \text{c1} \wedge \neg\, \text{c2} \leftarrow !c1_t,$
$c_f : \neg\, \text{c1} \wedge \text{c2} \leftarrow !c1_t\ ;\ !c2_f\}$

Figure 6.13: Plan generation rules (c_f) for boolean connectives

true and $c2$ false if $c1$ is false and $c2$ is true.

The generation rules for boolean connectives are shown in figures 6.12 and 6.13.

6.4.6 Rules for addition and deletion involving derived sets

Repair plans for propositions that are related to sets contain actions involving adding or removing entities from a set. However, sets can be derived from other sets by means of navigation or set operations supported in OCL such as *select*(), *reject*(), etc. Although adding an element to a basic single set (or removing it from the set) is a primitive action, adding or deleting elements from derived sets is not a primitive action. Instead, we model addition and deletion from derived sets as events, which are handled by additional plans. For example, in order to add the element x to $SE \stackrel{\text{def}}{=} S{\rightarrow}select(c)$ (previously denoted as the event $+(x, SE)$), we ensure both that x is in S, and that $c(x)$ is true.

The translation for this group of rules is defined as functions $\mathcal{Q}^+$ and $\mathcal{Q}^-$, each of which takes a derived set and returns a set of repair plans for adding an element to the set or removing an element from it respectively. Our repair plan generator covers the following cases of derived sets:

- **SE** $\stackrel{\text{def}}{=}$ **E.aend**: Returns a set of entities that are connected to E with regard to association end *aend*. Adding an entity x to this set is equivalent to connecting x with entity E with regard to association end *aend*. Removing an entity from this set is disconnecting the entity from E with regard to association end *aend*.

- **SE** $\stackrel{\text{def}}{=}$ **E.aend1.aend2**: Returns a set of entities that are connected (w.r.t. $aend2$) to entities that are connected to E with respect to $aend1$. There are two cases that we need to consider here:

  - $E.aend1$ leads to a single entity: adding an entity x to this set can be achieved by (i) adding x to $y.aend2$ and (ii) making $E.aend1 = y$ true for some y. Depending on the choice of y we may need to do either (i) – for y that E is currently connected to w.r.t. $aend1$, or (ii) – for y that is currently connected to x w.r.t. $aend2$, or both of them – for another existing entity y or a newly created one. Removing an entity x from this set is equivalent to disconnecting $E.aend1$ from x (w.r.t. $aend2$), or disconnecting E from $E.aend1$ (w.r.t. $aend1$).
  - $E.aend1$ leads to a set of entities: this is a shorthand notation of the *collect* operation, i.e. *E.aend1→collect(aend2)*. Adding an entity x to this set can be achieved by (i) adding x to $y.aend2$ and (ii) adding y to $E.aend1$ for some y. Depending on the choice of y we may need to do either (i) – for y that E is currently connected to w.r.t. $aend1$, or (ii) – for y that is currently connected to x w.r.t. $aend2$, or both of them – for another existing entity y or a newly created one. Removing an entity x from this set can be achieved by picking out all the entities in the set $E.aend1$ that connect to x w.r.t. $aend2$ and for each such entity either disconnecting it from x (w.r.t. $aend2$), or disconnecting it from E (w.r.t. $aend1$).

  It is also noted that this case can be generalised to multiple navigations (more than two), i.e. $E.aend1.aend2.aend3....$

- **SE** $\stackrel{\text{def}}{=}$ **S→select(c)**: A subset of SE containing all elements for which constraint c is true. Adding an entity x to this set can be achieved by different ways: if x is contained in S then we have to make constraint c true for x; if x is not in S then we need to add x to S and if c is false for x, we need to make c true for x. Removing an entity x from this set can be achieved by either making c false for x or removing x from S.

- **SE** $\stackrel{\text{def}}{=}$ **S→reject(c)**: A subset of SE containing all elements for which constraint c is false. Adding an entity x to this set can be achieved by different ways: if x is contained in S, then we have to make constraint c false for x; if x is not in S then we need to

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{E.aend}) =$
$\{+(x, SE) \leftarrow$ Connect E and x (w.r.t *aend*)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{E.aend1.aend2}) =$
$\{+(x, SE)$: isSingle(E.aend1) $\wedge$ E.aend1 = y
$\leftarrow$!+(x, y.aend2),
$+(x, SE)$: isSingle(E.aend1) $\wedge$ x $\in$ y.aend2
$\leftarrow$!$(E.aend1 = y)_t$,
$+(x, SE)$: isSingle(E.aend1) $\wedge$ y $\neq$ E.aend1
$\wedge$ x $\notin$ y.aend2 $\wedge$ y $\in$ Type(E.aend1)
$\leftarrow$!$(E.aend1 = y)_t$; !+(x, y.aend2),
$+(x, SE)$: isSingle(E.aend1)
$\leftarrow$ Create y : Type(E.aend1) ;
!$(E.aend1 = y)_t$; !+(x, y.aend2),
$+(x, SE)$: $\neg$ isSingle(E.aend1) $\wedge$ y $\in$ E.aend1
$\leftarrow$!+(x, y.aend2),
$+(x, SE)$: $\neg$ isSingle(E.aend1) $\wedge$ x $\in$ y.aend2
$\leftarrow$!+(y, E.aend1),
$+(x, SE)$: $\neg$ isSingle(E.aend1) $\wedge$ y $\notin$ E.aend1
$\wedge$ x $\notin$ y.aend2 $\wedge$ y $\in$ Type(E.aend1)
$\leftarrow$!+(y, E.aend1) ; !+(x, y.aend2),
$+(x, SE)$: $\neg$ isSingle(E.aend1)
$\leftarrow$ Create y : Type(E.aend1) ;
!+(y, E.aend1) ; !+(x, y.aend2)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S}\rightarrow\mathbf{select(c)}) =$
$\{ +(x, SE)$: x $\in$ S $\leftarrow$!c_t(x),
$+(x, SE)$: x $\notin$ S $\wedge$ c(x) $\leftarrow$!+(x, S),
$+(x, SE)$: x $\notin$ S $\wedge$ $\neg$ c(x) $\leftarrow$!+(x, S) ; !c_t(x)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S}\rightarrow\mathbf{excluding(e)}) =$
$\{+(x, SE)$: x $\notin$ S $\wedge$ x $\neq$ e $\leftarrow$!+(x, S)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1}\rightarrow\mathbf{union(S2)}) =$
$\{ +(x, SE)$: x $\notin$ S1 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\notin$ S2 $\leftarrow$!+(x, S2)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1}\rightarrow\mathbf{intersection(S2)}) =$
$\{ +(x, SE)$: x $\notin$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S2),
$+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S1) ; !+(x, S2)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1 - S2}) =$
$\{ +(x, SE)$: x $\notin$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!+(x, S1) ; !$-$(x, S2),
$+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!$-$(x, S2)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1}\rightarrow\mathbf{symmetricDifference(S2)})$
$=$
$\{ +(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S2),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!$-$(x, S1),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!$-$(x, S2)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S}\rightarrow\mathbf{including(e)}) =$
$\{+(x, SE)$: x $\notin$ S $\wedge$ x $\neq$ e $\leftarrow$!+(x, S)$\}$

$\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S}\rightarrow\mathbf{reject(c)}) =$
$\{ +(x, SE)$: x $\in$ S $\leftarrow$!c_f(x),
$+(x, SE)$: x $\notin$ S $\wedge$ c(x) $\leftarrow$!+(x, S) ; !c_f(x),
$+(x, SE)$: x $\notin$ S $\wedge$ $\neg$ c(x) $\leftarrow$!+(x, S)$\}$

Figure 6.14: Plan generation rules for basic propositions involving set addition

add x to S and if c is true for x, we need to make c false for x. Removing an entity x from this set can be achieved by either making c true for x or removing x from S.

- **SE** $\stackrel{\text{def}}{=}$ **S1→union(S2)**: The union of $SE1$ and $SE2$. Adding an entity x to this set can be achieved by adding x to either $S1$ or $S2$ if x does not belong to either of them. Removing an entity x from this set can be achieved by removing x from either $S1$ or $S2$ or both of them, depending on whether x is in $S1$, or $S2$, or both of them.

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **E.aend**) =
{$-(x, SE) \leftarrow$ Disconnect E and x (w.r.t *aend*)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **E.aend1.aend2**) =
{$-(x, SE)$: isSingle(E.aend1) ∧ E.aend1 = E1
← Disconnect E1 and x (w.r.t *aend*2),
$-(x, SE)$: isSingle(E.aend1) ∧ E.aend1 = E1
← Disconnect E and E1 (w.r.t *aend*1),
$-(x, SE)$: ¬ isSingle(E.aend1)
∧ E.aend1 = SE1
← for each y in SE1
if x ∈ y.aend2 then !aux(x, SE, y),
aux(x, SE, y) ← Disconnect y and x (w.r.t. *aend*2),
aux(x, SE, y) ← Disconnect E and y (w.r.t. *aend*1)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S→select(c)**) =
{ $-(x, SE) \leftarrow c_f$(x),
$-(x, SE) \leftarrow$!−(x, S)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S→reject(c)**) =
{ $-(x, SE) \leftarrow c_t$(x),
$-(x, SE) \leftarrow$!−(x, S)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S→excluding(e)**) =
{$-(x, SE)$: x ∈ S ∧ x ≠ e ← !−(x, S)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S1→union(S2)**) =
{ $-(x, SE)$: x ∈ S1 ∧ x ∈ S2 ← !−(x, S1) ; !-(x, S2),
$-(x, SE)$: x ∈ S1 ∧ x ∉ S2 ← !−(x, S1),
$-(x, SE)$: x ∉ S1 ∧ x ∈ S2 ← !−(x, S2)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S1→intersection(S2)**) =
{ $-(x, SE) \leftarrow$!−(x, S1),
$-(x, SE) \leftarrow$!−(x, S2)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S1 - S2**) =
{ $-(x, SE) \leftarrow$!−(x, S1),
$-(x, SE) \leftarrow$!+(x, S2)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S1→symmetricDifference(S2)**)
=
{ $-x$: x ∈ S1 ∧ x ∉ S2 ← !−(x, S1),
$-x$: x ∈ S1 ∧ x ∉ S2 ← !+(x, S2),
$-x$: x ∈ S2 ∧ x ∉ S1 ← !−(x, S2),
$-x$: x ∈ S2 ∧ x ∉ S1 ← !+(x, S1)}

$\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **S→including(e)**) =
{$-(x, SE)$: x ∈ S ∧ x ≠ e ← !−(x, S)}

Figure 6.15: Plan generation rules for basic propositions involving set deletion

- **SE** $\stackrel{\text{def}}{=}$ **S1→intersection(S2)**: The intersection of $SE1$ and $SE2$, i.e. the set of all elements that are in both $SE1$ and $SE2$. Adding an entity x to this set can be achieved

by adding x to either $S1$ (if x is not in $S1$ but in $S2$), or $S2$ (if x is not in $S2$ but in $S1$) or both of them (if x is not in either of $S1$ or $S2$). Removing an entity from this set is equivalent to removing it either from $S1$ or from $S2$.

- **SE** $\stackrel{\text{def}}{=}$ **S1 − S2**: The elements of set $SE1$, which are not in set $SE2$. Adding an entity x to this set can be done in different ways: if x is in $S2$ but not in $S1$ then we need to add x to $S1$ and remove it from $S2$; if x is neither in $S1$ nor $S2$ the we need to add x to $S1$; if x is in both $S1$ and $S2$, then we need to remove x from S2. Removing an entity from this set can be done by removing it from $S1$ or adding it to $S2$.

- **SE** $\stackrel{\text{def}}{=}$ **S1→symmetricDifference(S2)**: The set containing all the elements that are either in $SE1$ or $SE2$ but not in both. Adding an entity x to this set can be done in different ways: if x is in neither $S1$ nor $S2$ then we need to add x to either $S1$ or $S2$; if x is in both $S1$ and $S2$ then we need to remove it from either $S1$ or $S2$. Removing an entity from this set can be done by adding it to the set (either $S1$ or $S2$) that it does not belong to or removing it from the set that it does belong to.

- **SE** $\stackrel{\text{def}}{=}$ **S→including(e)**: The set containing all elements of SE plus e. Adding an entity to this set is achieved by adding it to the set S but only if this entity does not belong to S and is different from e. Removing an entity from this set is equivalent to removing it from the set S. If the entity to be added is e, then adding e to SE does not result in any effect and removing it from SE cannot be done.

- **SE** $\stackrel{\text{def}}{=}$ **S→excluding(e)**: The set containing all elements of SE without e. Adding an entity to this set is achieved by adding it to the set S but only if this entity does not belong to S and is different from e. Removing an entity from this set is equivalent to removing it from the set but only if this entity is in S and is different from e.

- **SE** $\stackrel{\text{def}}{=}$ **S→collect(aend)**: the shorthand of this notation is using multiple navigation, e.g. $E.aend1.aend2$. We have discussed this form of set expression earlier.

The generation rules for addition and deletion involving derived sets are shown in figures[10] 6.14 and 6.15.

[10]Function *isSingle*() tests if a given navigation results in a single entity. In addition, $E.aend1 = E1$ indicates that $E.aend1$ results in a single entity $E1$. Meanwhile, $E.aend1 = SE1$ indicates that $E.aend1$ leads to a set of entities $SE1$.

6.4.7 Discussion

In the previous sections, we define functions $\mathcal{P}_t$ and $\mathcal{P}_f$ that return a set of specific plans for a given constraint form. We now define a fuction $\mathcal{R}$ that takes an event (e.g. making a constraint true) and returns a repair plan set (unlike function $\mathcal{P}_t$ or $\mathcal{P}_f$ which takes a constraint not an event). More specifically, let $\mathcal{R}(c_t)$ and $\mathcal{R}(c_f)$ be the complete set of repair plans for the event of making constraint c true and false respectively (note that c_t and c_f are events). We then define a function $\mathcal{T}$ that takes a *constraint* and returns the complete repair plan set: $\mathcal{T}(c) = \mathcal{R}(c_t) \cup \mathcal{R}(c_f)$.

We now discuss what should be included in $\mathcal{R}(c_t)$ and $\mathcal{R}(c_f)$. Firstly, $\mathcal{R}(c_t)$ should contain $\mathcal{P}_t(c)$ (which is the specific plans for the constraint form) and similarly $\mathcal{R}(c_f)$ contains $\mathcal{P}_f(c)$.

In addition, a constraint may have sub-constraints and consequently fixing the parent constraint may lead to the need to repair sub-constraints. This dependency is reflected in the generated plans, e.g. the form of some repair plans for constraint $c \stackrel{\text{def}}{=}$ *SE→exists(c1)* contain an event posted to repair the sub-constraint $c1$. If a constraint has more than one sub-constraint then fixing one sub-constraint might conflict with the plan that repairs the other sub-constraint. Therefore, in order to guarantee that the generated repair plans always correctly fix the constraint, we include some top level plans. More specifically, $\mathcal{R}(c_t)$ includes $\{\text{fixC}_t : \text{c} \leftarrow \text{true}, \text{fixC}_t : \neg\, \text{c} \leftarrow !c_t \,;\, !\text{fixC}_t\}$. Similarly, $\mathcal{R}(c_f)$ includes $\{\text{fixC}_f : \neg\, \text{c} \leftarrow \text{true}, \text{fixC}_f : \text{c} \leftarrow !c_f \,;\, !\text{fixC}_f\}$.

Furthermore, our repair plans are structured in a hierarchical manner, i.e. a plan may post certain events which are handled by further plans. As a result, when we consider a complete set of repair plans for a specific constraint, we need to take into account plans that handle events. Events in our generated repair plans are generally in two forms: fixing a sub-constraint and adding/deleting an element to/from a derived set (see section 6.4.6). We denote $event(\mathcal{P}_t(c))$ as the set of events that are posted within all plans belonging in $\mathcal{P}_t(c)$, which may include events of the form $+(x, SE)$ and (x, SE). The complete set of repair plans $\mathcal{R}(c_t)$ for making constraint c true is defined as:

$$\mathcal{R}(c_t) = \mathcal{P}_t(c) \;\cup\; \{fixC_t : c \leftarrow true,\; fixC_t : \neg\; c \leftarrow !c_t;\; !fixC_t\} \;\cup \bigcup_{ev \in event(\mathcal{P}_t(c))} \mathcal{R}(ev)$$

and the complete set of repair plans $\mathcal{R}(c_f)$ for making constraint c false is as:

$$\mathcal{R}(c_f) = \mathcal{P}_f(c) \cup \{fixC_f : \neg\, c \leftarrow true,\ fixC_f : c \leftarrow !c_f;\ !fixC_f\} \cup \bigcup_{ev \in event(\mathcal{P}_f(c))} \mathcal{R}(ev)$$

The above definition is applied to the events representing the need to make a constraint true and false. With respect to the event representing addition of an entity e to a derived set SE, denoting $+(e, SE)$, we have the following definition.

$$\mathcal{R}(+(e, SE)) = \mathcal{Q}^{+}(SE) \cup \bigcup_{ev \in event(\mathcal{Q}^{+}(SE))} \mathcal{R}(ev)$$

Similarly to deletion to a derived set, we have the following definition:

$$\mathcal{R}(-(e, SE)) = \mathcal{Q}^{-}(SE) \cup \bigcup_{ev \in event(\mathcal{Q}^{-}(SE))} \mathcal{R}(ev)$$

In the next section, we illustrate how the repair plan generation rules are used to generate repair plans for a simple example.

6.5 Example

Now let us consider a simple example of how repair plans are generated for a consistency constraint in Prometheus (refer to the Prometheus metamodel presented in figure 4.12 on page 87, figure 4.13 on page 88, and figure 4.16 on page 92). The following constraint[11] specifies that any agent that handles a percept should contain at least one plan that is triggered by the percept.

Context Percept inv:

self.agent→forAll(a : Agent | a.plan→exists(pl : Plan | pl.percept→includes(self)))

We denote the above constraint as $c(self)$, and $c_t(self)$ is the event representing the need to make $c(self)$ true. We also define the following abbreviations:

c1(self, a) $\stackrel{\text{def}}{=}$ a.plan→exists(pl : Plan | pl.percept→includes(self))
c2(self, pl) $\stackrel{\text{def}}{=}$ pl.percept→includes(self)

[11] Here, association ends *agentsRespondingEntityReference*, *includedPlansEntityReference*, and *perceptsEntityReference* are written as *agent*, *plan*, and *percept* respectively for short.

Our repair plan generator produces the following repair plans for making constraint c true, since it has the form *SE→forAll(c)*.

c_t(self) ← for each a in self.agent if ¬ $c1_t$(self, a) then !c'_t(self, a) **(P1)**

c'_t(self, a) ← !−(a, self.agent)[12] **(P2)**

c'_t(self, a) ← !$c1_t$(self, a)[13] **(P3)**

For constraint $c1$ we generate the following plans, since the constraint is of the form $SE \rightarrow$ exists(c). In the rules of figure 6.8 "Type(SE)" denotes the type of SE's elements, in this case SE (which is *a.plan*) contains plans, and therefore in P5 the context condition requires that *pl* be an element of the set of all plans, denoted Set(Plan).

$c1_t$(self, a) : pl ∈ a.plan ← !$c2_t$(self, pl) **(P4)**

$c1_t$(self, a) : pl ∈ Set(Plan) ∧ pl ∉ a.plan ← !+(pl, a.plan) ; !$c2_t$(self, pl) **(P5)**

$c1_t$(self, a) ← Create pl : Plan ; !+(pl, a.plan) ; !$c2_t$(self, pl) **(P6)**

$c2_t$(self, pl) : c1(self, a) ∨ c2(self, pl) ← true **(P7)**

Similarly, for constraint $c2$ we generate the following plan.

$c2_t$(self, pl) ← !+(self, pl.percept) **(P8)**

Finally, by applying functions $\mathcal{Q}^+$ and $\mathcal{Q}^-$ we have the following plans for addition and deletion from derived sets.

−(a, self.agent) ← Disconnect *self* and *a* (w.r.t. *agent*) **(P9)**

+(pl, a.plan) ← Connect *a* and *pl* (w.r.t. *plan*) **(P10)**

+(self, pl.agent) ← Connect *pl* and *self* (w.r.t. *agent*) **(P11)**

Figure 6.16 represents the event-plan tree for fixing constraint c using the plans produced by our repair plan generator. At the root of the tree is the event of making the top constraint c true. The leaves of the tree are primitive repair actions. As can be seen, the top event $c_t(self)$ can trigger only one plan $P1$. From its definition, this plan can post multiple events $c'_t(self)$, as denoted with a * mark next to its node. It is noted that events and plans in this tree are event types and plan types, not instances. A single plan type can generate multiple

[12]Post an event to remove *a* from the set *self.agent*.

[13]Post an event to make $c1(self, a)$ true.

plan instances at runtime. For instance, plan P4 triggered by the event $c1_t$(self) has a free variable "pl" in its context conditions. At runtime, this variable is bound to different values, in this case it is bound to any entity contained in the set *a.plan*. For each value it is bound to, a corresponding plan instance is generated at runtime.

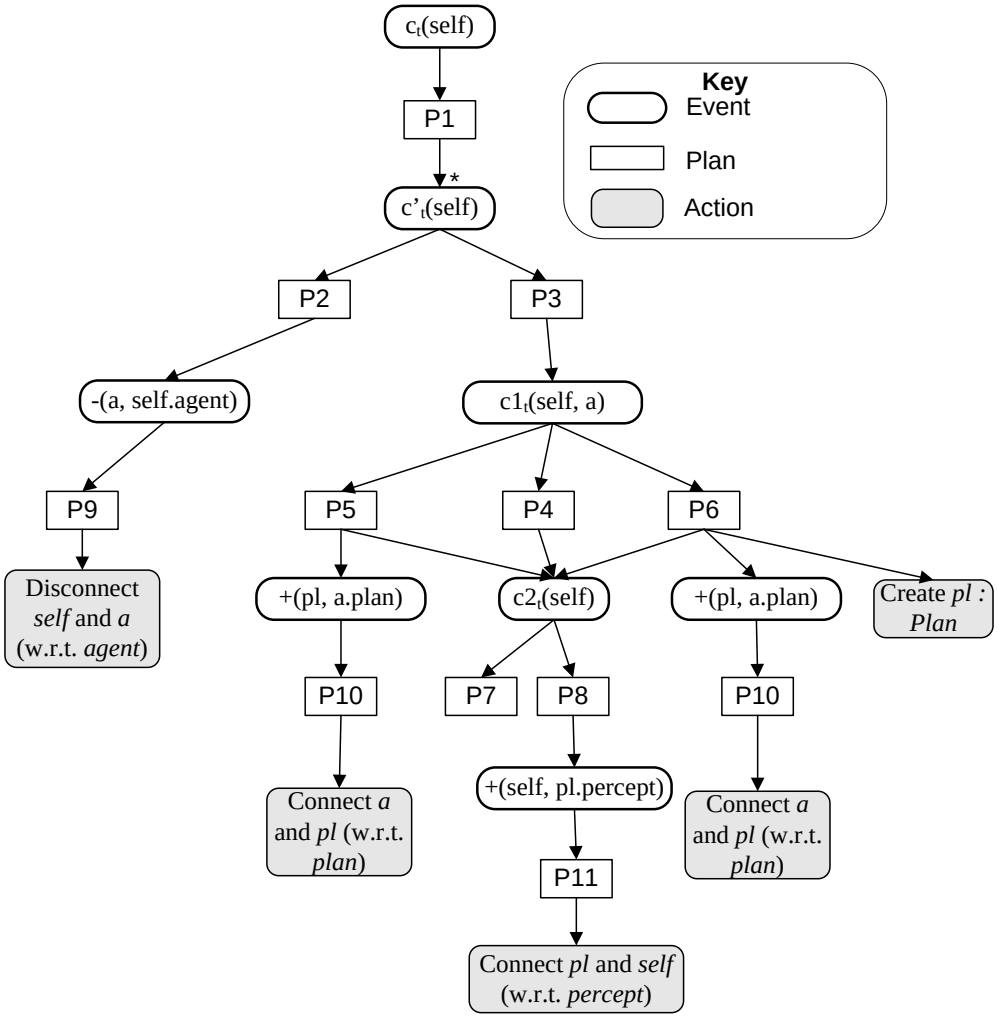


Figure 6.16: An event-plan tree for c_t(self)

By examining the event-plan tree, one is able to identify options to repair the top constraint. For instance, the event-plan tree in figure 6.16 gives the following repair options for fixing constraint $c(self)$, which are informally expressed as below:

1. For each agent handling the given percept *self*, if the agent does not contain at least one plan that is triggered by the percept, we do one of the following:

(a) Remove this agent from the percept, i.e. this agent no longer handles the percept (plans P2 and P9).

(b) Pick out one of the plans belonging to the agent, and make it handle the percept (P3, P4, P8 and P11).

(c) Pick one existing plan, add it to the agent (P3, P5, P10), and if the plan already handles the percept then do nothing (P7), else make the plan handle the percept (P8 and P11).

(d) Create a new plan, add it to the agent, and make it handle the percept (P3, P6, P10, P8, and P11).

An important question that can be raised here is whether the above repair options are correct and complete. In the next section, we attempt to prove the correctness and completeness of the generated repair plans.

6.6 Correctness and completeness

As mentioned earlier, the most important criteria of our repair plan generator are correctness and completeness. In this section, we prove, by induction, that the translation schema proposed above possesses these two properties.

A repair plan type contains a mixture of actions and subgoals (or alternatively called events). Such subgoals are achieved by executing further plans. At run time, a repair plan type (e.g. plan P in figure 6.17) is expanded into zero or more plan instances (e.g. plans P_1, P_2, ..., P_n) by solving the plan's context condition. When each plan instance (e.g. plan P_1) has all its subgoals resolved, it is then expanded into some number of action sequences (e.g. P_1^a, ..., P_1^m) by expanding each subgoal into a particular choice of plan (and doing this recursively for sub-subgoals and so on). We firstly define a sequence of repair actions that correctly fixes a given constraint.

Definition 2 (Correct sequence of repair actions)**.** *A sequence of actions S repairs constraint C in model M iff (a) C is violated in M; and (b) performing S on M yields a new model M'; and (c) C holds in M'.* ■

The above definition generalises to a set of constraints in the obvious way. Based on the definitions of repair actions in section 6.1, an important observation is that the preconditions

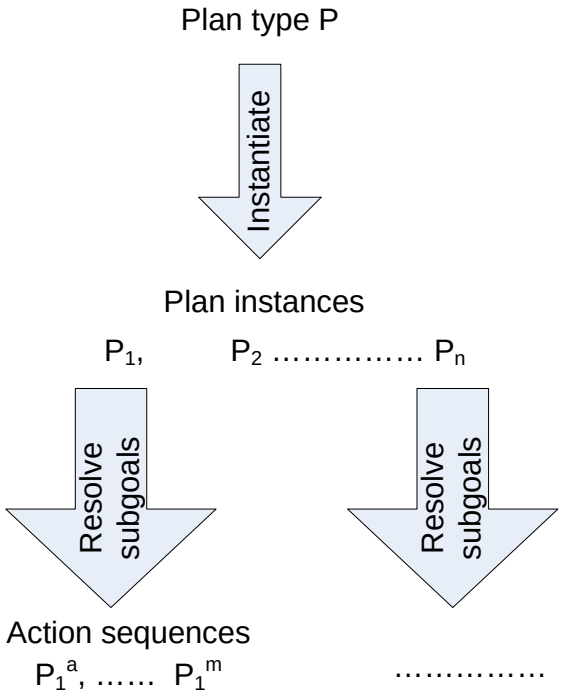


Figure 6.17: An example illustrating relationships between plan types, plan instances and action sequences

of these primitive actions are quite weak. This allows us to arbitrarily reorder a sequence of actions subject to the following conditions:

1. Creation of entities must remain before addition of relationships (i.e. connecting) between the entities;

2. If the sequence of actions has redundant pairs – an action that undoes the effects of an earlier action – then the pair cannot be swapped, but it can be simplified by deleting both of the two actions. For example, adding a relationship followed by deleting it can be replaced by simply doing nothing. Condition (2) is not needed if we assume that the sequence of actions being reordered is *non-redundant*, i.e. does not contain any redundant pairs.

Lemma 1 (Action sequence reordering)**.** *A non-redundant sequence of actions S can be arbitrarily reordered, so long as creation of entities precedes relating these entities, without affecting the overall effect of S.* ■

The permutation of actions within a particular action sequence may give a number of different action sequences which, according to lemma 1, give the same effect as long as they satisfy condition 1 mentioned earlier. For such action sequences, we are interested in their representative which is defined as below.

Definition 3 (Representative permutation of action sequences)**.** *A sequence of actions S is said to be a representative permutation of all action sequences that are derived by arbitrarily reordering it, subject to the constraint that creation of entities precedes operations on these entities.* ■

Based on the definition of a correct sequence of repair actions, we now define a definition of a *correct* repair plan.

Definition 4 (Correct repair plan)**.** *A repair plan type P correctly fixes a violated constraint C if and only if when P is instantiated and then resolved, it results only in correct sequences of actions for repairing C.* ■

As mentioned in chapter 3 (our change propagation framework) and discussed in detail in chapter 7, the selection of applicable repair plans is based on the notion of costs. As a result, when repair plans are generated at compile time, we only focus on plans that have no unnecessary steps in fixing a particular constraint. Such plans are considered as *minimal plans* which are defined as below. Similar to the way we define correct repair plans, we define minimal repair plans through the concept of minimal sequence of actions.

Definition 5 (Minimal sequence of repair actions)**.** *A correct sequence of actions S is minimal if removing actions from it always results in a sequence that no longer repairs C in M.* ■

Definition 6 (Minimal plan)**.** *A repair plan type P for fixing constraint C is said to be a minimal plan if and only if when P is instantiated and resolved, it results in only minimal sequences of actions for repairing C.* ■

Given the definitions of correct and minimal plans, a complete set of repair plans is defined below.

Definition 7 (Complete set of repair plans)**.** *A set of repair plan types $\mathcal{R}(C_t)$ for a constraint C is said to be complete if and only if, for any minimal (and correct) sequence of actions S*

for repairing C, a representative permutation of S can be obtained by instantiating and then resolving a plan in $\mathcal{R}(C_t)$. ■

Based on the above definitions, the correctness and completeness of our translation scheme for making a constraint true (i.e. $\mathcal{R}(C_t)$) are expressed by the following theorem. Note that a similar theorem can also be derived for the translation scheme for making a constraint false (i.e. $\mathcal{R}(C_f)$).

Theorem 1. *For any given OCL constraint* C *which is satisfiable*[14] *, the set of repair plans $\mathcal{R}(C_t)$ produced by the repair plan generator is correct and complete. That is, it generates*[15] *a representative permutation for each correct and minimal action sequence, and does not generate any incorrect action sequences, and all generated action sequences are minimal.*

Proof: We will prove, by induction, that the above theorem holds with respect to the translation schemas for making a constraint true, i.e. c_t. A similar proof can be used to prove its correctness for making a constraint false, i.e. c_f.

First of all, we will prove that theorem 1 holds for all the *basic* OCL constraints that we cover. This is relatively easy since repair plans are generated by considering all the possible ways in which a constraint can be false. In appendix A, we provide a proof of the theorem for all basic OCL constraints that are covered by our repair plan generator. We provide here a detailed proof for a typical example: c $\stackrel{\text{def}}{=}$ SE→includesAll(SE').

As can be seen in figure 6.8, the repair plan set $\mathcal{P}_t(c)$ for the given constraint c contains three main plan types: {$c_t \leftarrow$ for each x in (SE' − SE) !c'_t(x), c'_t(x) ← !−(x, SE'), c'_t(x) ← !+(x, SE)}. When these plan types are instantiated and resolved, we obtain a set of possible action sequences AS. We need to show that:

(a) for any $as \in AS$: (i) as is correct; and (ii) as is minimal.

(b) let as' be a correct and minimal action sequence, then as' must be either in AS or a representative permutation of an action sequence in AS.

In order to show (a), let $\{x_1, \ldots, x_n\} = SE' - SE$ (where we know that $n > 0$, else the constraint is already true), and, because it is a set, we also know that $x_i \neq x_j$ for $i \neq j$.

[14] Satisfiability here means that there exists a sequence of actions which makes the constraint true. We do not actually check for satisfiability – we assume that the user will not ask for impossible constraints, and the system does actually detect failure in these cases.

[15] In this context, plan generation means instantiation of plans followed by resolving subgoal choices.

From the above plan type definitions, any action sequence as in AS contains actions that for each x_i either remove it from SE' or add it to SE.

The above constraint can be written as:

$$c \stackrel{\text{def}}{=} \forall\, x \in SE' \bullet x \in SE$$

Assume that c is violated, i.e. $\neg$ c is true, expressed as follows.

$$\begin{aligned} \neg\, c &\stackrel{\text{def}}{=} \neg\, (\forall\, x \in SE' \bullet x \in SE) \\ &\stackrel{\text{def}}{=} \exists\, x \in SE' \bullet x \notin SE \end{aligned}$$

Now, $SE' - SE$ is the set of x_i for which the condition $x_i \in SE' \wedge x_i \notin SE$ holds. Therefore, constraint c is violated if and only if there is an x_i which is in SE' and not in SE. Hence to fix c (minimally) must ensure that no such x_i exists. We do this by finding all x_i for which the condition holds (which is given by $SE' - SE$) and for each x_i making the condition false, which can be done by either deleting it from SE' (to make $x_i \in SE'$ false) or adding it to SE (to make $x_i \notin SE$ false). This matches exactly with an action sequence as in AS.

Depending on the nature of the sets SE and SE', there are further plans contained in $\mathcal{R}(c_t)$, and we need to reason by induction. Let us consider as an example the case where $SE \stackrel{\text{def}}{=} E.aend$. By applying the rules described in figure 6.14, there is another plan in $\mathcal{R}(c_t)$: $+(x, E.aend) \leftarrow$ Connect E to x (w.r.t $aend$). $E.aend$ is the set of entities that connect to E with respect to association end $aend$. Therefore, the only action for adding an entity x to the set $E.aend$ is connecting E to x with respect to $aend$ (see the taxonomy of repair actions in section 6.1).

Now, in order to prove (b), let as' be an action sequence that is both correct (for fixing c) and minimal. We argue that as' must be either one of the action sequences in AS or a representative permutation of an action sequence in AS. Let x_i be in $(SE' - SE)$ then, as argued earlier, as' must somehow remove x_i from $(SE' - SE)$, and this can be done in exactly two (minimal) ways: removing it from SE', or adding it to SE.

Furthermore, as' must do this for each x_i in $(SE' - SE)$ – any x_i that is not removed

leaves a violation of the constraint. Finally, removing each of the x_i from $(SE' - SE)$ is sufficient to repair the constraint, and therefore as' does not do anything else (since it is minimal). Thus we have argued that as' must follow the pattern of action sequences in AS.

An OCL constraint is ultimately a combination (*and*, *or*, *not*, *xor* and *implies*) of basic constraints. We have proved that theorem 1 holds for all basic constraints. We now use that to prove, by induction, that theorem 1 holds for the basic connectives: *and*, *or*, and *not*. The other connections (*xor* and *implies*) can be derived from the basic ones. Below is a proof for the *or* expression. For the others, please refer to appendix A.

For c $\stackrel{\text{def}}{=}$ c1 or c2, assume that theorem 1 holds for $\mathcal{R}(c1_t)$ and $\mathcal{R}(c2_t)$, i.e. both of them are correct and complete sets. Now we need to prove that it also holds for $\mathcal{R}(c_t)$.

According to figure 6.12, we have:

$$\mathcal{P}_t(c) \quad = \quad \{c_t : \neg\ c1 \leftarrow !c1_t, c_t : \neg\ c2 \leftarrow !c2_t\}$$

and we also have:

$$\mathcal{R}(c_t) = \mathcal{P}_t(c) \cup \mathcal{R}(c1_t) \cup \mathcal{R}(c2_t) \cup \ldots$$

Because of our induction assumption $\mathcal{R}(c1_t)$ is correct and complete, i.e. it contains plans that correctly (and minimally) fix $c1$, and similarly for $\mathcal{R}(c2_t)$ and $c2$. Therefore, plan c_t : $\neg$ c1 $\leftarrow$ $!c1_t$ is able to repair c1 and plan c_t : $\neg$ c2 $\leftarrow$ $!c2_t$ is able to repair c2. Since the constraint c holds if either of $c1$ or $c2$ holds, any plan that is able to fix $c1$ or $c2$ can fix c. As a result, we can conclude that $\mathcal{R}(c_t)$ contains plans that correctly fix c. These plans are also minimal because they do not contain redundant repair actions. For instance, plan c_t : $\neg$ c1 $\leftarrow$ $!c1_t$ fixes only $c1$ when $c1$ is false, which just sufficiently repairs c without the need to fix $c2$.

We have proved that $\mathcal{R}(c_t)$ contains correct and minimal repair plans for c. Now we prove the completeness of the set $\mathcal{R}(c_t)$. Assume that there is a minimal plan P that fixes c and does not belong to $\mathcal{R}(c_t)$. Plan P must fix either $c1$ or $c2$ (but not both, otherwise it is not minimal), and without loss of generality we assume that P aims to fix $c1$. Therefore, plan P is also the minimal plan for fixing $c1$, which results in, due to the induction assumption,

that P belongs to $\mathcal{R}(c1_t)$. Since $\mathcal{R}(c_t)$ contains $\mathcal{R}(c1_t)$, P also belongs to $\mathcal{R}(c_t)$, which contradicts our previous assumption. Hence, there does not exist any minimal plan P that fixes c and does not belong to $\mathcal{R}(c_t)$, i.e. the set $\mathcal{R}(c_t)$ is complete.

The induction proof above shows that the generated repair plans of an *or* constraint correctly fix the constraint. The proofs for all other constraint forms are presented in appendix A. One interesting case that should be noted involves constraint forms that have repair plans for fixing sub-constraints conflicting with each other. For instance, for $c \stackrel{\text{def}}{=} c1 \textit{ and } c2$ the generated repair plans are[16]:

$$\begin{aligned}\mathcal{R}_t(c) \quad = \quad & \{\mathit{fixC}_t : c \leftarrow \mathit{true}, \mathit{fixC}_t : \neg\ c \leftarrow !c_t;\ !\mathit{fixC}_t\} \cup \mathcal{R}(c1_t) \cup \mathcal{R}(c2_t) \cup \\ & \{c_t : \neg\ c1 \wedge c2 \leftarrow !c1_t, c_t : \neg\ c2 \wedge c1 \leftarrow !c2_t, c_t : \neg\ c1 \wedge \neg\ c2 \leftarrow !c1_t\}\end{aligned}$$

Assume that c is false because $c1$ is true and $c2$ is false, then fixC_t calls the plan aiming to fix $c2$, i.e. $c_t : \neg\ \mathrm{c2} \wedge \mathrm{c1} \leftarrow !c2_t$. However, this plan may make $c1$ become false, which results in c still being false. Since fixC_t is called recursively until c becomes true, the plan aiming to fix $c1$ is called, i.e. $c_t : \neg\ \mathrm{c1} \wedge \mathrm{c2} \leftarrow !c1_t$. However, this plan may also make $c2$ false, in which case the plan aiming to fix $c2$ is called and this may continue as a loop. In general, if it is not possible to make both $c1$ and $c2$ true at the same time, i.e. every plan that fixes $c1$ violates $c2$ and vice versa, then c is not satisfiable. If c is satisfiable, then there exists a repair plan that is able to fix c. We are able to prove (refer to appendix A) that such a repair plan can be generated by $\mathcal{R}$. Furthermore, in this case our plan selection mechanism (discussed in chapter 7) will find terminating repair plans by favoring them over any other repair plans that cause an infinite loop.

Overall, we can conclude that our generated repair plans for a constraint correctly fix it if the constraint is satisfiable. Our cost algorithm (described in chapter 7) is able to detect infinite loops caused by conflict between repair plans fixing sub-constraints. ■

6.7 Related work

There has been a range of work on automatic repair of constraints in the area of databases. In these approaches, production rules are often used to represent various ways of fixing a given violated constraint. A rule premise contains conditions and if the premise is satisfied, actions

[16]The full proof for this constraint form is presented in appendix A.

specified in the consequent of the rule can be executed. One key difference between their work and ours is that we generate abstract, structured, repair plans that are instantiated at runtime.

The work in the area of databases focuses on integrity constraint maintenance [Mayol and Teniente, 1999], i.e. making changes to transactions or databases to recreate a state of integrity. While approaches in this area share a common goal, which is avoiding costly transaction roll-back by fixing constraint violations before committing, they differ in several aspects such as the expressiveness of the constraints that they allow, the degree of user interaction, and the assurance that can be made about repairs. We now describe some of the most relevant work in the area of relational, object-oriented, active and deductive databases. Firstly, Urban et al. [1992] proposed an approach to generate repair actions for a given constraint in the context of active integrity maintenance of an object-oriented database[17]. Their approach also works automatically from a declarative specification but it can only handle simple boolean constraints in Skolem normal form.

The approach described in [Ceri et al., 1994] also supports automatic generation of production rules but focuses on integrity enforcement for relational databases. Moreover, their rule generator takes constraints in more complicated forms, i.e. existential and universal quantifications. In addition, they also allow the user to modify the generated rules, as well as the order on constraints to be checked and the order on rules to be executed. In our framework, the user is also able to modify generated repair plans but they do not need to take care of the order of constraints or rules to be executed. In fact, as we show in chapter 7, with regard to the specific problem that we are trying to solve, the order in which constraints are fixed does not affect the outcome. Similar to our approach, they also involve user intervention in selecting repair actions but they use a low-level text mechanism and no tool support is discussed. The work described in [Gertz and Lipeck, 1997] covers a similar class of constraints (as in [Ceri et al., 1994]) but they specifically raise the issue of user involvement in the process of resolving constraint violation, i.e. allow the user to choose between possible alternative repairs that reflect the user's intention or the application requirements. They propose several repair strategies for reducing the set of repair actions presented to the user for selection such as minimizing the number of changes necessary to the database or the

[17]The consistency of a database can be specified using passive and/or active integrity constraints. While passive constraints are used for simply checking validity, active constraints ensure database consistency by adding or deleting data [Cacace et al., 1990].

transactions triggering the repairs. These strategies are, however, analytical and generic (i.e. not developed based on domain knowledge), which differ from our repair plans. Nonetheless, we also share the same concern as one of their repair strategies in terms of only including repair plans which make changes necessary to the application model.

There have also been work on deriving repair actions in deductive databases. For example, Moerkotte and Lockemann [1991] proposed a system that generates actions from closed, range-restricted first order logic formulae. Since their repair algorithm depends on the rules of the database and the closed-world assumption, it can automatically find repairs for violated existential formulae without user intervention. Such assumptions are not available in the case of design models and thus a mechanism of interactively querying the user, in some cases, to select a specific repair action for violated constraints is needed.

Our work is also related to the approach for repairing inconsistent distributed and structured documents proposed in [Nentwich et al., 2003]. Constraints between distributed documents are expressed in xlinkit, a combination of first order logic with XPath expressions. Their framework is powerful in terms of automatically deriving a set of repair actions from the constraints. In addition, they use abstract repair actions as a way to as a way to reasonably represent the potentially large number of concrete ways of resolving constraint violations. However, they consider only a single change, and do not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. As a result, their approach does not explicitly address the cascading nature of change propagation. Our repair plan generator is, however, built on top of the intuition and ideas proposed in their approach.

6.8 Chapter summary

In this chapter we have described in detail one of the key and novel components of our change propagation framework: a repair plan generator. The key idea of this generator is automatically producing repair plans for a given OCL constraint. Its availability significantly facilitates the task of the repair administrator in terms of developing and managing repair plans which is the means of propagating changes. In this chapter, we have formally defined repair actions using the Z notation. We have also discussed several key issues and solutions involving automatically generating repair plans by analysing OCL constraints. In addition, we completed the abstract syntax of repair plans previously mentioned (in chapter 3). The

key part of this chapter is the set of generation rules for common OCL invariants. Finally, we have proved that the repair plans generated following our rules are correct, minimal and complete.

As shown in most of plan generation rules, there are usually multiple repair plans for a given constraint. This leads to another key question that needs to be answered which is how to select between different applicable repair plans to fix a given constraint violation. Resolving this issue is the focus of the next chapter.

Chapter 7

Plan Selection

In the previous chapter, we have discussed the repair plan generator, an important component of our framework, which is responsible for producing repair plan types for consistency constraints. These plan types are instantiated at run time to become plan instances. The availability of the repair plan generator provides a significant step in our effort to improve the automation of change propagation. However, another important issue that needs to be answered as part of the effort is how to select between different applicable (repair) plan instances to fix a given constraint violation. This chapter serves to address this issue. We begin with a discussion of the challenges in dealing with the problem of repair plan selection and propose our approach of using a notion of repair plan cost (section 7.1). We also explain why this cost calculation mechanism is suitable to reflect the cascading nature of change propagation. In section 7.2 we present a series of formal definitions for the cost of actions, plans, constraints and (sub)goals. In section 7.3 we then explore some properties of the definitions that enable us to derive cost algorithms. We present two versions of the cost algorithms: one involving an exhaustive search and the other with pruning capabilities (section 7.4). We also analyse and discuss the complexity of the algorithms.

7.1 Issues and solutions in repair plan selection

There can be multiple applicable repair plans for resolving a given inconsistency. For instance, an inconsistency concerning a naming mismatch between a message in a sequence diagram and the operations in the message's receiver class (as discussed in chapter 5) can be resolved in different ways: either changing the message's name or changing the name of one of the

operations. Choosing between those different possible repair plans can depend on various factors. For example, assume that the inconsistency is caused because the designer has renamed the message so that its name matches with a corresponding state transition. In this case, renaming an operation is possibly more preferable from the designer's perspective.

The above example has shown that the cause of inconsistency can play a role in determining which repair plans should be chosen. Other elements that can influence repair plan selection include user dependent factors such as the designer's style and preferences. Formulating all of those factors is not generally feasible [Egyed and Wile, 2006], thus a completely automated mechanism for selecting repair options is not appropriate. On the other hand, it is not desirable for our change propagation mechanism to involve a substantial amount of user interaction. We therefore develop a semi-automated mechanism for repair option selection: options that are considered infeasible and costly (explained below) are automatically filtered out, and the user is presented with a set of "quality" (i.e. low cost) repair options for selection. It is noted that we assume that cheapest repair options are the "best" options although this may not be necessarily the case in practice. More investigation on this issue is part of future work as discussed in chapter 10.

We view inconsistency resolution in the context of change propagation. It means that we have to consider the side-effects of a repair plan on other constraints. They can be negative effects, i.e. breaking a constraint, or positive effects, i.e. resolving a violated constraint. As a result, the selection of repair plans is also dependent on their side-effects.

Our approach to the above issues relating to plan selection is to define a suitable notion of repair plan cost. Generally, the cost of a repair plan indicates the number of repair actions that are needed to fix a given constraint violation. There are several important properties related to our notion of repair plan cost.

- It can reflect the user preferences in terms of biasing repair plans that have cheaper cost. This implies that we can minimise the user interaction during the change propagation process. It reduces the load on the user by presenting them with a subset of possible repair options, namely the cheapest ones.

- It provides a simple mechanism for the user to adjust the change propagation process. For example, if the user wishes to bias the change propagation process towards adding more information then he/she may assign lower costs to actions that create new entities or add entities, and higher costs to actions that delete entities. Another example is the

case where the user wants a particular type of artefact (e.g. class diagrams) to take precedence over others (e.g. statechart diagrams). He/she can assign higher costs to actions that change class diagrams and lower costs to actions that modify statechart diagrams.

- It provides an effective way of accounting for the side-effects of a repair plan. More specifically, side-effects can be measurable in terms of costs and can be included in the cost of a repair plan. Negative side-effects increase the cost of a repair plan whilst positive side-effects reduce its cost. By modelling side-effects as costs, we can also effectively eliminate infeasible cyclic repair plans because such plans would result in an infinite cost.

In the next section, we discuss the cost definition of repair plans in more details.

7.2 Cost definition

In this section, we give equations that define the cost of fixing a given constraint using repair plans. The notion of cost that we use is abstract: it can be viewed as counting the number of primitive actions (creation, deletion, connection, disconnection and modification) involved in a given repair plan. For example, if repair plan P_1 involves 5 connections and repair plan P_2 involves 3 connections then we view P_2 as being cheaper. In order to compare "apples and oranges", e.g. if P_3 involves two connections and a creation, we assume that each primitive action type is assigned a numerical cost (its "basic cost"), for instance creation may have an assigned cost of 5 and connection a cost of 3. These numbers do not correspond to any real cost, and are simply used to compare different action types.

Let us define some preliminary concepts and terminology.

Definition 8 (Action cost)**.** *The cost of an action is defined by a function* cost *that maps an action to a natural number.* cost(A) *is the user-defined basic cost associated with the repair action type* A *(i.e. creation, deletion, connection, disconnection, and modification).* ■

A constraint that does not hold with regard to a model is said to be violated, and can be fixed by executing a repair plan. Based on the abstract syntax of repair plans, we can view a repair plan instance as comprising primitive repair actions and subgoals (also referred to as events).

Definition 9 (Primitive actions of a repair plan[1])**.** *The primitive actions of a repair plan are defined by a function $\mathcal{A}$ that maps a repair plan to a set of primitive repair actions (i.e. creation, deletion, connection, disconnection, and modification). In other words, $\mathcal{A}(P)$ returns the set of the repair actions in plan P.* ■

The cumulative costs of those primitive repair actions in a repair plan form the plan's basic cost. Calculating the basic costs of a repair plan is straightforward and is done by summing the costs of all actions in the plan.

Definition 10 (Basic cost)**.** *The basic cost of a plan is defined by a function* basicCost *that maps a plan to a natural number. The basic cost of a plan is the sum of the costs of all actions of the plan.*

$$basicCost(P) \quad = \quad \sum_{A \in \mathcal{A}(P)} cost(A)$$

■

Aside from primitive actions, a repair plan also contains subgoals/events which in most cases correspond to resolving violated sub-constraints or adding/removing concerning sets.

Definition 11 (Subgoals of a repair plan)**.** *The subgoals of a repair plan are defined by a function $\mathcal{G}$ that maps a repair plan to a set of subgoals. In other words, $\mathcal{G}(P)$ returns the set of subgoals of plan P.* ■

Similar to primitive actions, the costs of subgoals in a repair plan form the subgoal cost of the plan. Calculating a plans's subgoal cost is more complicated than its basic cost because one needs to work out the cost of each subgoal in the plan. We will define the cost of a goal later.

Definition 12 (Subgoal cost)**.** *The subgoal cost of a plan is defined by a function* subGoal-Cost *that maps a plan to a natural number. The subgoal cost of a plan is the sum of the costs of all subgoals in the plan.*

$$subGoalCost(P) \quad = \quad \sum_{G \in \mathcal{G}(P)} cost(G)$$

[1] In this section, repair plan instances are referred to as repair plans, and constraint instances are referred to as constraints.

■

We now define the main cost of a plan in terms of the costs of its basic actions (basicCost) and the cost of its subgoals (subGoalCost). It is noted that the main cost is different from the cost of a repair plan which also takes into account the costs of fixing other violated constraints – which we will discuss next.

Definition 13 (Main cost)**.** *The main cost of a plan is defined by a function* mainCost *that maps a plan to a natural number. The main cost of a plan is the sum of the plan's basic cost and its subgoal cost.*

$$\begin{aligned} mainCost(P) &= basicCost(P) + subGoalCost(P) \\ &= \sum_{A \in \mathcal{A}(P)} cost(A) + \sum_{G \in \mathcal{G}(P)} cost(G) \end{aligned}$$

■

As previously discussed, repairing a constraint often has side-effects. Such side-effects include both introduction of new constraint violations and the repair of existing violations. Therefore, it is very important in the context of change propagation to consider a group of constraints together when fixing a single constraint. We name this group of constraints the *repair scope*. When a repair plan completes fixing a constraint, the other constraints in the repair scope are checked and any violated constraints are then repaired. A global repair scope involves all constraints whilst a local one contains constraints related to certain entities in the model.

Definition 14 (Repair scope of a repair plan)**.** *The repair scope of a repair plan is defined by a function $\mathcal{S}$ that maps a repair plan to a set of constraints. In other words, $\mathcal{S}(P)$ returns a set of constraints, which is called the repair scope, of plan P.* ■

A repair scope is related to the "distance" of change propagation, i.e. how far changes are propagated in a design model. For instance, if one wants to propagate changes from design artefacts to specification artefacts, one needs to consider a repair scope that contains constraints which affect model elements in those artefacts. In our framework, we allow the repair administrator to group constraints into different repair scopes, which enables them to

limit the propagation to certain constraints or model entities. Normally the repair scope is set initially (typically to be global) and then is not changed.

Our cost calculation takes into account the side-effects of a plan with respect to its repair scope by having a concept of a (repair) scope cost for each repair plan. If a repair plan positively contributes towards its repair scope, e.g. does not break any constraint and/or fixes some other violated constraints, then its scope cost will be low. On the other hand, if a repair plan has a negative impact on its repair scope, e.g. breaks some other constraints, then its scope cost will be high. Hence, the scope cost of a repair plan is measured through the number of violated constraints and the costs of fixing them after the plan's execution. In order to calculate the scope cost of a repair plan, one needs to execute the plan and work out which constraints in the plan's repair scope become violated and which constraints are no longer violated.

Definition 15 ((Repair) scope cost)**.** *The (repair) scope cost of a plan is defined by a function* scopeCost *that maps a plan to a natural number. The scope cost is the cost of repairing all (violated) constraints in the plan's repair scope after the execution of the plan*[2]*.*

$$scopeCost(P) \quad = \quad \sum_{C \in \mathcal{S}(P)} cost(C)$$

■

We now define the cost of a plan in terms of its main cost and repair scope cost. It is noted that a repair scope is only meaningful to top-level plans, i.e. those that fix the top-level constraints. As a result, when calculating the cost of a plan we need to know if it is a top-level plan or not.

Definition 16 (Plan cost)**.** *The cost of a plan is defined by a function that maps a plan to a natural number. If a plan is to fix a top level constraint, the plan's cost is equal to the sum of its main cost and its repair scope cost. If a plan is to achieve a subgoal, the plan's cost is equal to its main cost.*

$$cost(P) = \begin{cases} mainCost(P) + scopeCost(P) & \text{if } P \text{ is top-level plan} \\ mainCost(P) & \text{otherwise} \end{cases}$$

[2]Since we will define $cost(C) = 0$ if the constraint C is not violated we simply sum over the cost of all constraints in $\mathcal{S}(P)$.

■

We now provide equations that define the cost of constraints and goals/events. A goal/event can be either fixing a top-level constraint, a sub-constraint or adding/removing elements to/from derived sets. There are usually several applicable plan instances to repair a constraint violation or to achieve a goal.

Definition 17 (Applicable plans)**.** *The applicable repair plans of a constraint is defined by the function $\mathcal{CP}$ that maps a constraint to a set of plan instances that can be used to fix the constraint. The applicable plans for achieving a goal is defined by the function $\mathcal{GP}$ that maps a goal to a set of plan instances that can be used to accomplish a goal.* ■

The best plan, which is selected for execution, is the one with minimum cost. Hence the cost of repairing a constraint is the cost of the cheapest repair plan instance.

Definition 18 (Constraint cost)**.** *The cost of a constraint*[3] *is defined by a function* cost *that maps a constraint to a natural number. The cost of fixing constraint C is equal to the cost of the best applicable repair plan instance with regard to C. If there are no applicable repair plans, the cost of C is undetermined*[4] *. The cost of fixing an unviolated constraint is 0.*

$$cost(C) = \begin{cases} 0 & \text{if } C \text{ unviolated} \\ min\ \{cost(P) \mid P \in \mathcal{CP}(C)\} & \text{otherwise} \end{cases}$$

■

Similarly, we define the cost of a plan achieving a goal. It is noted that the cost of a goal of fixing a constraint is equal to the cost of the constraint.

Definition 19 (Goal cost)**.** *The cost of a goal is defined by a function* cost *that maps a goal to a natural number. The cost of achieving a goal is the cost of the cheapest available repair plan.*

$$cost(G) = min\ \{cost(P) \mid P \in \mathcal{GP}(G)\}$$

■

[3]The cost of a constraint (instance) is relative to a particular state of the model.
[4]In this case we treat the cost as infinite.

We have provided equations to define the cost of plans, constraints, and goals. In order to illustrate how those definitions are applied in reality, we now calculate the costs for the following example.

7.2.1 Example

Assume that a model contains one agent $a1$ which plays one role $r1$ and handles one percept $p1$. Note that role $r1$ is not set to deal with percept $p1$. In addition, role $r1$ is also the only role in the model. It is noted that we use this simple and artificial example so that it is easy to demonstrate the pruning that is discussed later.

We now consider a repair scope that contains the following constraint:

- For any agent a and percept p that is handled by the agent, there is at least one of the roles played by agent a that is able to deal with percept p. This constraint is written in OCL in the context of the Prometheus metamodel (refer to figure 4.10 on page 81 and figure 4.13 on page 88) as below[5].
  c1(a, p) $\stackrel{\text{def}}{=}$ *a.role→exists(r : Role | r.percept→includes(p))*

We define constraint *c1'(r, p)* $\stackrel{\text{def}}{=}$ *r.percept→includes(p)*, which is a sub-constraint of $c1$, i.e. constraint $c1$ can be written as *c1(a, p)* $\stackrel{\text{def}}{=}$ *a.role→exists(r : Role | c1'(r, p))*

By applying the repair plan generation rules as previously discussed in chapter 6, we derive the following repair plan types for making constraint $c1$ true:

$c1_t$(a, p) : r ∈ a.role ← !$c1'_t$(r, p)
$c1_t$(a, p) : r ∈ Set(Role) ∧ r ∉ a.role ← !+(r, a.role); if ¬ c1'(r, p) !$c1'_t$(r, p)
$c1_t$(a, p) ← Create r : Role ; !+(r, a.role) ; if ¬ c1'(r, p) !$c1'_t$(r, p)
+(r, a.role) ← Connect a to r (w.r.t. *role*)
$c1'_t$(r, p) ← !+(p, r.percept)
+(p, r.percept) ← Connect r to p (w.r.t. *percept*)

Given the existing model, constraint $c1(a1, p1)$ is violated. We use the above repair plan types to generate the following repair plan instances for fixing this constraint. It is noted that $a1.role = \{r1\}$ and the set of roles currently has only $r1$, i.e. $Set(Role) = \{r1\}$. As a

[5]Here, association ends *perceptsEntityReference* and *rolesEntityReference* are written as *percept* and *role* respectively for short.

result, the second plan type to handle event $c1_t(a1, p1)$ does not yield any instances because its context condition has no solutions, i.e. no existing roles belonging to $a1.role$.

$c1_t$(a1, p1) ← !$c1'_t$(r1, p1) **(P1)**

$c1_t$(a1, p1) ← Create r2 : Role ; !+(r2, a1.role) ; !$c1'_t$(r2, p1) **(P2)**

+(r2, a1.role) ← Connect a1 to r2 (w.r.t. role) **(P3)**

$c1'_t$(r1, p1) ← !+(p1, r1.percept) **(P4)**

$c1'_t$(r2, p1) ← !+(p1, r2.percept) **(P5)**

+(p1, r1.percept) ← Connect r1 to p1 (w.r.t. percept) **(P6)**

+(p1, r2.percept) ← Connect r2 to p1 (w.r.t. percept) **(P7)**

There are two applicable repair plans for making constraint $c1(a1, p1)$ true: $P1$ and $P2$. Therefore, the cost of fixing this constraint is the minimum of $cost(P1)$ and $cost(P2)$.

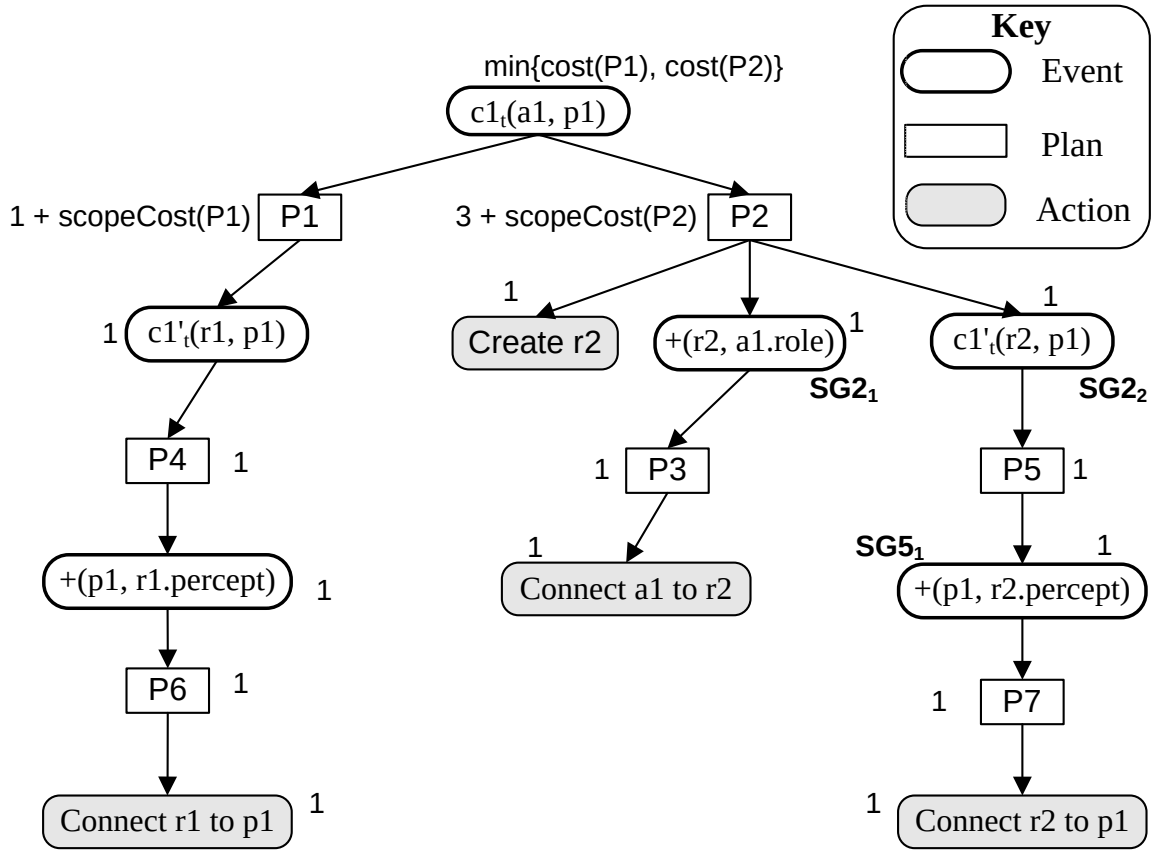


Figure 7.1: An example of cost calculation for constraint $c1(a1, p1)$

Figure 7.1[6] shows the goal-plan tree of repair plans for fixing constraint $c1(a1, p1$ and how the costs are calculated. We assume a basic cost of 1 for all action types used in this example. Basically, the costs of plan $P2$ are calculated as.

$$\text{cost}(P2) = \text{mainCost}(P2) + \text{scopeCost}(P2)$$

Where,

$$\begin{aligned}
\text{mainCost}(P2) &= \text{basicCost}(P2) + \text{subGoalCost}(P2) \\
&= \text{cost}(\mathit{Creation}) + \text{cost}(SG2_1) + \text{cost}(SG2_2) \\
&= \text{cost}(\mathit{Creation}) + \text{cost}(P3) + \text{cost}(P5) \\
&= \text{cost}(\mathit{Creation}) + \text{cost}(\mathit{Connection}) + \text{cost}(SG5_1) \\
&= \text{cost}(\mathit{Creation}) + \text{cost}(\mathit{Connection}) + \text{cost}(P7) \\
&= \text{cost}(\mathit{Creation}) + \text{cost}(\mathit{Connection}) + \text{cost}(\mathit{Connection}) \\
&= 3
\end{aligned}$$

Therefore, the cost of plan $P2$ is: $\text{cost}(P2) = 3 + \text{scopeCost}(P2)$
Similarly, the cost of plan $P1$ is:

$$\begin{aligned}
\text{cost}(P1) &= \text{mainCost}(P1) + \text{scopeCost}(P1) \\
&= 1 + \text{scopeCost}(P1)
\end{aligned}$$

Although the main cost of $P1$ is smaller than the main cost of $P2$, P1 may not be the plan chosen to fix $c1$. This is because its total cost, which includes the scope cost, is not guaranteed to be less than the cost of plan $P2$. In other words, we need to know the scope cost of each plan before making the decision of which plan to choose for fixing constraint $c1$. For instance, if the repair scope contains only constraint $c1$, then the scope costs of plans $P1$ and $P2$ are 0. Therefore, the cost of $P1$ is 1, which is smaller than the cost of $P2$ (i.e. 3) and consequently $P1$ is chosen to repair $c1$. However, consider the case where the repair

[6]The number placed next to each node is its corresponding cost.

scope contains another constraint $c2$ requiring that every agent needs to play at least 2 roles. In this case, plan $P2$ has a scope cost of 0, since after its execution, none of the constraints in the repair scope are violated ($c2$ is not violated since agent $a1$ now plays two roles $r1$ and $r2$). On the other hand, the scope cost of plan $P1$ is not 0 but is equal to the cost of constraint $c2$ (it is violated because agent $a1$ plays only 1 role $r1$). By performing a similar calculation, one should derive the cost of constraint $c2$ is 2 (1 for creating a new role and 1 for connecting it with agent $a1$). Therefore, the cost of plan $P1$ would be 3, which is equal to the cost of $P2$. As a result, with regard to this repair scope both plans could be chosen to fix $c1$.

7.3 Properties of the cost definitions

Having defined the costs of repairing violated constraints, we now consider a number of properties of these definitions.

Firstly, a repair plan type contains a mixture of actions and subgoals. Such subgoals are achieved by performing further actions. Therefore, a repair plan instance, which has all its subgoals resolved, is ultimately a sequence of actions that the user needs to perform in order to fix a particular violated constraint. By viewing an expanded repair instance in terms of the corresponding action sequence, we are able to ease our proof of the lemma and theorem presented ahead.

Definition 2 (on page 155) defines a sequence of actions that is able to repair a given constraint. We now define the cost of such an action sequence.

Definition 20 (Repair action sequence cost)**.** *The cost of a sequence of repair action S is defined by a function cost that maps an action sequence to a natural number. The cost of an action sequence is the sum of the costs of all actions of the sequence.*

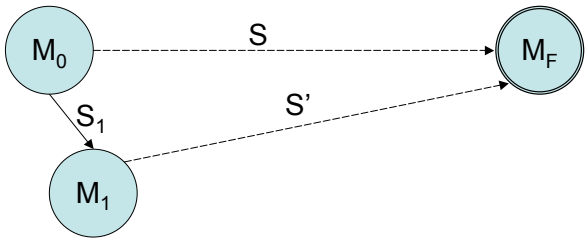
$$cost(S) \quad = \quad \sum_{A \in S} cost(A)$$

We have previously defined a minimal sequence of repair actions (refer to definition 5 on page 157). It is noted that minimality implies non-redundancy, i.e. all minimal plans are non-redundant. In addition, the definition of minimality generalises to a set of constraints in the obvious way. This definition gives us the following important lemma and theorem.

Lemma 2. *Let M_0 be a model in which the constraints C_i are violated. Let S be a minimal sequence of actions for repairing all the constraints C_i in M_0. Then for a given constraint, say (without loss of generality) C_1, there exists at least one sequence of actions S' which is obtained by removing some number (possibly zero) of actions from S such that S' repairs C_1 in M_0 and is minimal.*

Proof: S repairs C_1 in M_0, but may contain actions that are unnecessary for repairing C_1 (since S also repairs other constraints C_i where $i \neq 1$). We construct S' by simply removing these unnecessary actions, resulting in a minimal S'. ■

Theorem 2. *Let M_0 be a model where some number of constraints C_i are violated and let S be a minimal (and hence non-redundant) sequence of actions that repairs the C_i in M_0, yielding model M_F:*



Then (by lemma 2) for any of the given constraints, say (without loss of generality) C_1, there exists a minimal action sequence S_1 that repairs C_1 in M_0 yielding M_1. Furthermore, there then exists a minimal (and hence non-redundant) action sequence S' that takes us from M_1 to M_F where $cost(S) = cost(S_1) + cost(S')$.

Proof: We construct S' and S_1 from S as follows. We form S_1 by removing actions from S to yield a minimal S_1 for repairing C_1 in M_0 (using lemma 2). The actions that are not removed from S are the remainder, S'. We can view the sequence S_1 followed by S' as being a reordering of S, and by lemma 1 (on page 156) it has the same effect as S, i.e. results in M_F. Since S_1 followed by S' has the same actions as S it must have the same cost.

Now we prove, by contradiction, that S' is minimal. Assume that S' is not minimal. Then there exists a sub-sequence S'' of S' which is obtained by removing unnecessary actions, such that S'' correctly repairs all constraints C_i where $i \neq 1$. Now consider the action sequence $AS = S_1 \ ; \ S''$. This action sequence, from state M_0, correctly repairs C_1 using S_1 (resulting in state M_1), and then proceeds (using S'') to correctly repair the remaining constraints

C_i ($i \neq 1$). In other words, AS correctly repairs constraints C_i from state M_0. But by construction AS is a sub-sequence of S which is known to be minimal. This contradicts our assumption, and hence shows that S' is indeed minimal as desired. ■

By applying this theorem repeatedly, on C_1, then C_2, etc. we can show that in order to repair a set of violated constraints we can consider a single constraint at a time, in an arbitrary order, with no loss of generality. Furthermore, since the repair plans are complete (theorem 1 on page 158), the action sequence S_1 can be generated by instantiating the repair plan set. This strong result is only possible because the actions we consider have limited preconditions (refer to the formal definitions of repair actions in section 6.1 on 126), allowing them to be reordered fairly freely. This result holds for our case studies involving Prometheus and UML design models because we use those same types of repair actions in both cases. The reordering may not be necessarily possible if different actions with stronger preconditions are included. A specific corollary is that, considered as a planning domain, our actions do not allow for a Sussman anomaly situation[7] [Russell and Norvig, 2003, page 414] to exist.

7.4 Cost calculation algorithms

In the previous section, we have defined how a repair plan's cost is calculated. We now give algorithms that calculate this cost. The algorithms operate with plan-goal trees, where a goal has as children the plans that can be used to achieve it (denoted $\mathcal{P}(G)$) and a plan has as children its sub-goals (denoted $\mathcal{G}(P)$). Each plan node stores the plan's basic cost (*basicCost*, initially the basic cost of the plan), other costs (*dynamicCost*, initially 0), a boolean value indicating whether the node is a leaf (*isLeaf*, initially false), and a queue of its sub-goals (*subGoalQueue*, initially empty). Each goal node stores a list of best (i.e. least cost) plan(s) (*bestPlans*, initially empty) that achieve the goal.

Before we present the algorithm, we discuss a tree transformation that the algorithm uses. When considering the alternative ways of dealing with a given (sub)goal the algorithm considers the available plans and selects the cheapest. In doing so, it needs to consider the future: what will happen after the goal is handled. We do this by transforming the tree so that the "future" is pushed down into the tree beneath the current goal. Specifically, when we consider a goal that has a future (i.e. a parent plan with non-empty sub-goals) we copy

[7] Sussman's anomaly is well-known as an example of a planning problem in which a side-effect of establishing one (sub)goal is to deny another (sub)goal regardless of the order in which one tries to achieve the (sub)goals.

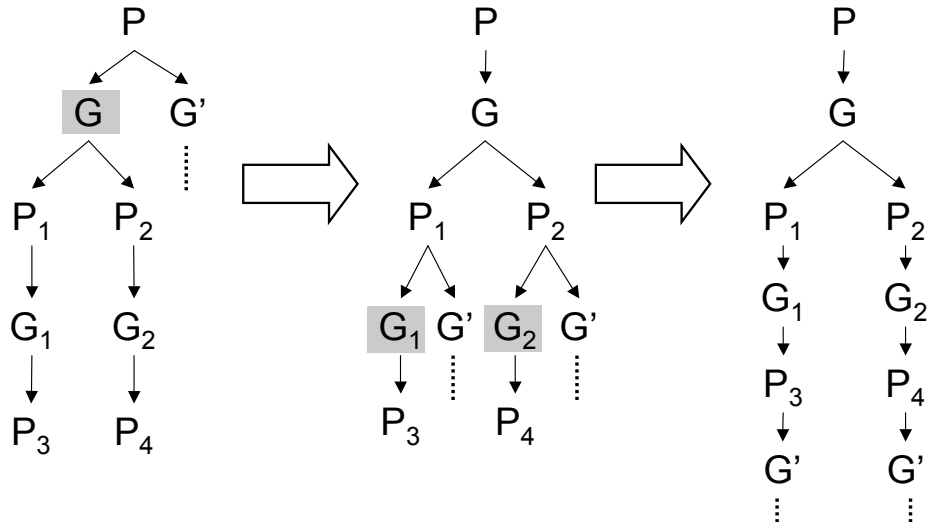


Figure 7.2: Tree Transformation

the sub-goals of the parent plan to the sub-goals of the children plans. As seen in figure 7.2, when considering two different plans P_1 and P_2 that handle goal G, we need to take into account the effects of each plan on achieving goal G'. As a result, the tree is transformed in which goal G' is pushed down to become the last subgoal of plans P_1 and P_2. Similarly, for the same reasons, goal G continues being pushed down to be a subgoal of plans P_3, P_4, and so on. It is noted that this technique is not generally applicable since a plan (e.g. P_1) may not have information accessible to its parent plan (e.g. P). In our case, however, the design model is the only information needed and is made globally accessible to all plans.

7.4.1 Initial algorithms

The algorithm presented in figure 7.3 computes the cost of a plan according to the equations in the previous section. Since we assume that *basicCost* is already computed (by simply summing the costs of primitive actions in a plan), the algorithm only needs to work out the plan's subgoal costs and repair scope costs (see definition 16). These costs are stored in *dynamicCost* which is initially set to 0, and is progressively incremented with the costs of sub-goals and of violated constraints in the repair scope.

The algorithm selects each sub-goal in turn (lines 2 and 3) and adds the cost of any violated constraints onto the dynamic cost (line 6). If the plan node has children (i.e. violated

```
   function cost(P)
1     P.isLeaf ← true
2     while P.subGoalQueue is NOT empty do
3        dequeue subGoal from P.subGoalQueue
4        if the constraint associated with subGoal is violated then
5           P.isLeaf ← false
6           P.dynamicCost ← P.dynamicCost + cost(subGoal, P)
7        end if
8     end while
9     if P.isLeaf = true then
10       local violatedSubGoals ← get-scope-violated-constraints()
11       if violatedSubGoals is NOT empty then
12          get a random violatedSubGoal from violatedSubGoals
13          enqueue violatedSubGoal into P.subGoalQueue
14          return cost(P)
15       end if
16    end if
17    return P.dynamicCost + P.basicCost
```

Figure 7.3: Calculating Plan Node Cost (No Pruning)

constraints[8]) then we are done, since the scope cost will be calculated in those children. On the other hand, if this plan node has no children (*isLeaf* = *true*, line 9) then we check for violated constraints in the repair scope (lines 10 and 11), and if there are any, we select one of the violated constraints (line 12), add it to the queue (line 13), and recursively call $cost(P)$ to compute its cost (line 14).

The algorithm in figure 7.4 calculates the cost of a goal node (see definition 19) by considering the possible plans and looking for the cheapest one. We first retrieve a list of applicable plans for the goal (line 2). We then iterate through the list of plans (line 4) and calculate the cost for each of them (line 9). When a plan that is cheaper than the previous best is found, the previous best plan(s) are replaced with the new plan (lines 10-13). When a plan is found that is as good as the current best, it is added to the list of best plan(s) (lines 14-15).

The algorithm uses look-ahead and simulates the application of the plans. Line 5 executes the plan currently being considered by (a) updating the model with the effects of the plan's actions, and (b) adding the plan's sub-goals to the tree. In order to be able to consider

[8]Note that when we encounter a violated constraint we note that the plan node is not a leaf (line 5).

```
   function cost(G, ParentPlan)
1     local bestCost ← +∞
2     local planList ← get-repair-plans(G)
3     G.bestPlans ← empty
4     for each plan P in planList do
5        execute plan P (in simulation)
6        if ParentPlan is not null then
7           copy all ParentPlan.subgoals to the end of P.subgoals
8        end if
9        local c ← cost(P)
10       if c < bestCost then
11          bestCost ← c
12          clear G.bestPlans
13          add P to G.bestPlans
14       else if c = bestCost then
15          add P to G.bestPlans
16       end if
17       unexecute plan P (in simulation)
18    end for
19    if ParentPlan is not null then
20       ParentPlan.subgoals ← empty
21    and if
22    return bestCost
```

Figure 7.4: Calculating Goal Node Cost (No Pruning)

alternative plans we need to undo the effects of the plan's execution on the model, and this is done by line 17. This is implemented by logging changes to the model, allowing these changes to be rolled back.

Lines 6-8 and 19-21 implement the tree transformation discussed earlier: the sub-goals of the parent plan (excluding the current sub-goal) are added to the end of the sub-goals of each plan P (lines 6-8). Once this has been done for all plans, we remove the sub-goals from the parent (lines 19-21).

7.4.2 Advanced algorithms with pruning capabilities

The algorithms given in figures 7.3 and 7.4 implement the definitions given in section 7.2, but they search the whole goal-plan tree. This is inefficient, and may lead to non-termination, since the tree may be infinite. We therefore modify the algorithms by adding loop checking,

```
   function cost(P)
1     P.isLeaf ← true
2     while P.subGoalQueue is NOT empty do
3        dequeue subGoal from P.subGoalQueue
*4       if P.β = +∞ and subGoal is in history then
*5          clear history
*6          return +∞
*7       end if
8        if the constraint associated with subGoal is violated then
9           P.isLeaf ← false
*10         local threshold = P.σ + lowerBoundCost(subGoal) +
                                  P.basicCost + P.dynamicCost
*11         if threshold > P.β then
*12            return threshold
*13         end if
*14         subGoal.σ ← P.σ + P.dynamicCost + P.basicCost
*15         subGoal.β ← P.β
16          P.dynamicCost ← P.dynamicCost + cost(subGoal, P)
17       end if
18    end while
19    if P.isLeaf = true then
20       violatedSubGoals ← get-scope-violated-constraints()
21       if violatedSubGoals is NOT empty then
22          get a random violatedSubGoal from violatedSubGoals
23          enqueue violatedSubGoal into P.subGoalQueue
24          return cost(P)
25       end if
26    end if
*27   P.β ← min(P.β, P.σ + P.dynamicCost + P.basicCost)
28    return P.dynamicCost + P.basicCost

   function lowerBoundCost(G)
*1    local planList ← get-repair-plans(G)
*2    local lowerBound ← +∞
*3    for each plan P in planList do
*4       if P.basicCost < lowerBound then
*5          lowerBound ← P.basicCost
*6       end if
*7    end for
*8    return lowerBound
```

Figure 7.5: Calculating Plan Node Cost (Pruning)

and a form of pruning. We add to each goal/plan node two values[9]: β (initially $+\infty$) - the least cost of fixing all constraints in the repair scope, and σ (initially 0) - the (accumulative) cost of everything *above* the current node. In figures 7.5 and 7.6 lines that are new (relative to figures 7.3 and 7.4) are marked with "*".

```
   function cost(G, ParentPlan)
1     local bestCost ← +∞
2     local planList ← get-repair-plans(G)
3     G.bestPlans ← empty
*4    remove plans in planList that have basic cost greater than G.β
*5    sort plans in planList based on their basic action costs
*6    if G.β = +∞ then
*7       add G into history
*8    end if
9     for each plan P in planList do
*10      P.β ← G.β
*11      P.σ ← G.σ
12       execute plan P (in simulation)
13       if ParentPlan is not null then
14          copy all ParentPlan.subgoals to the end of P.subgoals
15       end if
16       c ← cost(P)
17       if c < bestCost then
18          bestCost ← c
19          clear G.bestPlans
20          add P to G.bestPlans
*21         G.β ← P.β
22       else if c = bestCost then
23          add P to G.bestPlans
24       end if
25       unexecute plan P (in simulation)
26    end for
27    if ParentPlan is not null then
28       ParentPlan.subgoals ← empty
29    end if
30    return bestCost
```

Figure 7.6: Calculating Goal Node Cost (Pruning)

Computing the cost of a plan is done by the algorithm in figure 7.5. We use a pruning

[9]We use "β" since we do the β part of a classical $\alpha - \beta$ pruning. We do not do the α part because we have a min-sum tree, rather than a min-max tree.

mechanism, where we establish a threshold in order to avoid exploring alternatives that are more expensive than known solutions. The threshold is calculated (line 10 of figure 7.5) based on the current accumulative cost σ, the plan cost ($P.basicCost$ and $P.dynamicCost$) and the lower bound cost, which is an estimate of the minimum cost of achieving a (sub-)goal (lines 1-8 in the bottom of figure 7.5).

The algorithm presented has two forms of loop detection. The first is a consequence of pruning: if one way of repairing the violated constraints has been found then the algorithm cannot fall into a loop, since the cost of a loop increases, and eventually the algorithm detects this and prunes the loop. The second loop detection mechanism is used when no repair plan has yet been found (lines 4-7 of figure 7.5 and lines 6-8 of figure 7.6). This second mechanism keeps track of all of the sub-goals encountered, and if the same goal (i.e. instantiated by the same constraint type and model entity instances) is seen again, corresponding to the fact that a constraint has become violated and is being fixed again, then we have a loop and we terminate with infinite cost. These two mechanisms together are quite effective at detecting and avoiding non-termination, and were successful in all of our testing and experiments. However, these mechanisms are not perfect: it is possible to envisage pathological cases[10] where the algorithm fails to terminate. However, these pathological cases appear to be unlikely, and indeed, we believe (but have not proven) that they are ruled out by the plan generation mechanism.

The two values β and σ are passed from each parent goal/plan node down to its child plan/goal nodes (lines 14-15 in figure 7.5 and lines 10-11 in 7.6). Line 14 in figure 7.5 shows that σ is in fact an accumulative cost: we accumulate the cost of the current node in σ. When a plan cost is resolved, the total cost so far (i.e. the cost of the plan as well as σ, the cost of the path from the root of the tree to the current node) is compared against the current β to see if it needs to be updated (line 27 in figure 7.5). If at any point the total cost for a plan (*threshold*) exceeds β then we prune (lines 11-13 of figure 7.5). We also prune in the (admittedly unlikely) case that a plan's basic cost by itself exceeds β (line 4 of figure 7.6). Once a best plan for a goal is found, the goal's β is also updated with the plan's β

[10]For example, suppose that we have a hypothetical domain-specific constraint c that each agent type has a "supervisor" agent type, and there is only one applicable plan for repairing this constraint (if it is violated with regard to a particular agent): creating a new agent and making the new agent the supervisor of the existing agent. Also, assume that the model initially has only one agent a_1. Since a_1 needs a supervisor, we create agent a_2 and make it the supervisor for a_1. However, agent a_2 now needs a supervisor, which leads to the creation of agent a_3, and so on. Since the goal instance is different in each case (e.g. $c(a_1), c(a_2), etc.$) the same exact constraint never appears twice in the history, so the termination checker cannot detect the loop.

(line 21 of figure 7.6). Line 5 of figure 7.6 implements a heuristic that considers plans with cheaper basic cost first.

7.4.3 Example

We now give an example to illustrate how the algorithms work. We use the same example as in section 7.2.1 with the assumption that the repair scope contains only one constraint, i.e. $c1$. Figure 7.7 shows a plan-goal tree with the values associated with each node. Firstly, it can be seen that we apply the tree transformation to the subgoal $c1'(r2, p1)$ by moving it and all of its children to be a subgoal of plan $P3$.

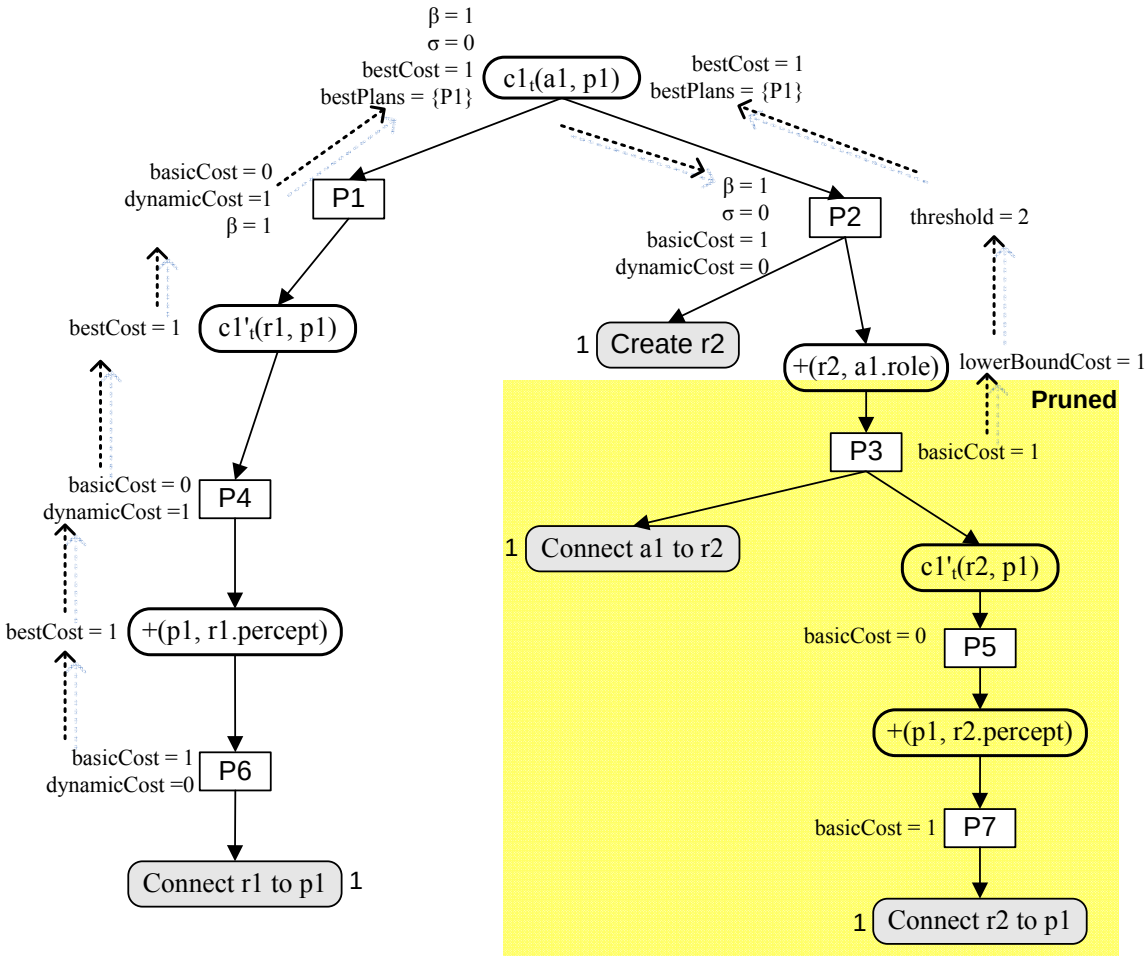


Figure 7.7: An example of cost calculation for constraint $c1(a1, p1)$

On the leftmost branch of the tree, values are propagated from the leaf to the root. For

example, the *basicCost* of plan $P6$ is 1 (because it has only one connection action) and its dynamic cost is 0 (because it has no subgoals), resulting in its total cost being 1. Since subgoal $+(p1, r1.percept)$ has only one plan $P6$, its *bestCost* is clearly equal to the cost of the plan, i.e. 1. Similarly, after the branch starting with plan $P1$ is fully explored, the values at the root node $c1_t(a1, p1)$ are: $\beta = 1$, $\sigma = 0$, $bestCost = 1$, and $bestPlans = \{P1\}$. After that, the algorithm begins calculating the cost of nodes on the branch starting with plan $P2$. The values of β (1) and σ (0) are propagated from the root node to the plan node $P2$ (corresponding to lines 10 and 11 in figure 7.6). Plan $P2$ then executes (line 12) and its cost is calculated (line 16). This leads to line 10 of figure 7.5 where the *threshold* variable is calculated. Note that the *lowerBoundCost* of subgoal $+(r2, a1.role)$ is 1 because the subgoal has only one plan, i.e. $P3$ that has a basic cost of 1 (since it has only one connection action). As a result, the *threshold* has the value of 2, which is greater than the value of β. Therefore, the tree is pruned here (lines 11 and 12 of figure 7.5) and the cost of plan $P2$ is returned with the value of 2. This cost is greater than the current best cost of the top goal $c1_t t(a1, p1)$, therefore the best cost is 1 and the best plan is $P1$.

7.4.4 Complexity analysis

The computational complexity of the algorithm depends on the cost of checking all constraints in the repair scope, as well as the number of child nodes each (non leaf) node has (N), the depth of the plan-goal tree (D), and the size of the application design model, i.e. the number of model elements (M). Based on empirical evidence [Egyed, 2006], we assume the cost of checking a single constraint to be constant. More specifically, we assume constraints are not likely to be enormously complex. In addition, from a complexity analysis point of view we can determine the largest constraint size, c, and use this as the constant. As a result, checking all constraints in the repair scope is basically proportional to the size of the repair scope (E). Furthermore, although we do not include the tree transformation in the analysis (lines 6-8 of figure 7.3), we note that the transformation may create duplicate nodes, which consequently creates duplicated work.

For each plan node, we need to execute the plan, i.e. perform update actions on the application design model (line 5 of figure 7.4) and unexecute the plan, i.e. undo what was previously done to the model (line 17 of figure 7.4). As executing/unexecuting a plan involve actions that affect only a small part of the application model, we assume it to be constant,

i.e. not depending on the size of the application model. Moreover, for each plan node we also need to calculate the cost of all of the plan's subgoals (lines 2-8 of figure 7.3). As the maximum number of subgoals is N, the work to be done for each plan node is $O(N)$. Finally, we may also need to perform constraint evaluation to identify violated constraints in the repair scope (line 10 of figure 7.3). As a result, the work to be done for each plan node may also include evaluating constraints in the repair scope, i.e. $O(E)$. Overall, the worst-case complexity of each plan node is, therefore, $O(N + E)$.

For each goal node, we need to get a list of applicable repair plan instances (line 2 of figure 7.4), calculating the cost of each of these plans (line 9 of figure 7.4), work out the cheapest plans and store them with the goal node (line 10-16 of figure 7.4). As can be seen in chapter 6, the repair plans that generate many instances are the ones which have a context condition of the form $x \in SE$, where SE is set of existing entities in the model. As a result, time taken to generate applicable repair instances from repair plan types tends to be proportional to the number of model elements, i.e. M. The work to calculate the cost of each plan is already considered above. Finally, the time taken of sorting and storing cheapest repair plans is proportional to $N \log N$. As a result, the work to be done for each goal node is roughly $O(N \log N + M)$.

Since the number of nodes is roughly $O(N^D)$ this gives an overall computational complexity of $O(N^D \times (N \log N + M + N + E))$. Because $N < N \log N$, the overall computational complexity is $O(N^D \times (N \log N + M + E))$.

The algorithms are exponential in the number of violated constraints in a repair scope due to an extensive look-ahead planning. In practice, this issue can be dealt with by limiting the number of constraints in a repair scope. A smaller repair scope will reduce the number of plan and goal nodes to be explored, hence improving the performance of the algorithms. The trade-off is, however, that the user needs to carefully select which constraints should be included in the repair scope. In addition, after all changes are made the design may still be inconsistent with respect to other constraints that are not taken into account in the repair scope. Furthermore, as noted earlier, the number of constraints in the repair scope is related to the distance of change propagation. As a result, limiting the number of constraints in the repair scope would affect how far changes are propagated in the design. Finally, an extreme solution is considering a single constraint at a time, instead of the whole repair scope (i.e. equivalent to a repair scope of size 1). The algorithms is roughly linear in this case. The

trade-off of this solution is that side-effects are not taken into account and consequently the cascading nature of change propagation is ignored.

In chapter 9, we will discuss an evaluation to assess the algorithm's performance empirically, and its results show that despite exponential worse-case complexity, the algorithm is viable for small to medium examples. In addition, although pruning was not taken into account in the analysis, it was assessed empirically in chapter 9.

7.5 Related work

The cost calculation algorithm can be seen as a form of reasoning about an agent's plans, albeit in a special setting. There has been previous work on investigating the interaction between plans either within a single agent or between different agents in a multi-agent system (e.g. [Clement and Durfee, 1999; Thangarajah et al., 2002]). There are some similarities between this work and ours, for example, a plan's cost can be viewed as its resource consumption and the fact that fixing one constraint can partially/totally repair other constraints can be seen as positive interaction between plans. However, there are several major differences between their work and ours. First of all, the selection between applicable plans is not controllable. Secondly, the algorithms of [Thangarajah et al., 2002] rely on a finite plan-goal tree, whereas our algorithm does not require a complete tree, rather, the search tree is pruned as soon as cheaper plans are identified.

The issue of calculating the cost of a plan or a goal in the context of existing plans has been previously addressed in [Horty and Pollack, 2001]. The aim of their work is to determine whether an agent should adopt a new goal. They estimate the cost (with a range) rather than calculate the exact cost like our work. In addition, the plans which they consider contain only primitive actions, and they require complete plans. We also found that it is not easy to adopt their approach to deal with selecting between alternative plans, as opposed to deciding whether to adopt a goal.

Surprisingly, the specific problem of selecting between applicable plans in BDI agents has not received much attention. One particular work that tackles this issue is presented in [Dasgupta and Ghose, 2006]. They extend AgentSpeak(L) to deal with intention selection in BDI agents. They also use a lookahead technique to work out the potential cost of a plan and choose the best plan to execute, and their plan representation is also hierarchical. However, there are several differences between their work and ours. Firstly, they impose a limit on

the plan-goal tree by giving the depth of the tree as an input to their algorithm. Secondly, they assume that the environment changes rapidly and expect the worst case scenarios when looking ahead. In the domain that we are interested in, the environment is static so we always choose the least cost plans. Finally, they do not consider costs in the context of existing plans.

Our process for computing cost — performing lookahead over an and-or tree — clearly resembles a planning problem, and it can be viewed as such, with a few rather specific requirements. Firstly, because we have repair plans we want to use an HTN (Hierarchical Task Network) planner. Secondly, we want to collect the set of all best (cheapest) plans, so we need a planner that supports a notion of plan cost, and is able to collect all cheapest plans. Finally, because we have a large, potentially infinite, search space, we want a planner that does pruning and loop detection. Unfortunately, we do not know of any planner that meets all three requirements.

Perhaps the closest is SHOP2 [Nau et al., 2003] which is an HTN planner that supports collecting all best plans and that does branch and bound pruning. However, SHOP2 does not do loop detection, and although it provides iterative deepening, which can be used to avoid looping, this does not return the cheapest solution(s), as required. We encoded a UML design[11] and associated constraints and repair plans using SHOP2. Our experiments have shown that SHOP2 gives the same results as our cost calculation if it terminates, but that it is susceptible to looping, and that SHOP2 is slightly slower than our Java implementation (0.172 seconds vs. 0.157 seconds[12]).

Optimal planning is an area clearly related to our work. Optimal planning extends traditional plans by continuing to search for the optimal plan instead of stopping when the first plan is found. There has been a range of work which applies decision theory to classical planning to deal with optimal solutions [Blythe, 1999; Williamson, 1994]. They are similar in terms of integrating a utility model to the plans. One significant trend in extending classical planning in dealing with optimization is to use heuristics to guide the search [Ephrati et al., 1996]. Each node in the state space is evaluated based on a heuristic function. Heuristic planners then try to search the state space by looking at the most promising branches first. Note that calculating heuristic functions can be computationally

[11]The video-on-demand system [Egyed, 2006], and see http://peace.snu.ac.kr/dhkim/java/MPEG/

[12]On a Windows XP PC with a 1.73Ghz CPU and 1GB RAM, using Java v.1.5.0_06 and SHOP2 v1.3 running with GNU CLISP v2.3 for Windows.

expensive, often in proportion to the accuracy of the heuristics.

7.6 Chapter summary

In this chapter, we have discussed in detail the cost calculation component, an important part of our framework. This component deals with the issue of choosing between possible repair plans. We have raised the challenges of repair plan selection and argued that our approach using a notion of cost is effective. In addition, we have presented detailed formal definitions of costs for repair plans, constraints and other related entities. We have also proved an important property derived from those definitions, which indicates that regardless of the order of constraints being fixed, the total cost of fixing all of them is the same. Furthermore, we presented algorithms that perform cost calculation based on those definitions and properties, and we have applied pruning techniques to improve the performance of the algorithms.

This chapter also concludes the details of our change propagation framework. In the next section, we describe an implementation of the framework and in chapter 9 we discuss an evaluation of our approach.

Chapter 8

Implementation

In the previous chapters, we have presented a theoretical foundation for our approach to change propagation, and discussed its details. In this chapter, a proof-of-concept tool support is presented for demonstrating how our approach works in practice. The prototype tool is called Change Propagation Assistant and is integrated with the Prometheus Design Tool (PDT)[1], a modelling tool that supports the Prometheus methodology for designing agent-based systems. First, we present an overview of the architecture of Change Propagation Assistant (section 8.1). Next, a brief description of the Dresden OCL2 Toolkit, which provides an important platform for our tool, is presented (section 8.2). We then explain two major components of our tool, Repair Plan Generator and Change Propagation Engine, in sections 8.3 and 8.4 respectively.

8.1 Architectural overview

Figure 8.1 shows the architecture of the Change Propagation Assistant. The tool relies on the Dresden OCL2 Toolkit[2] platform in terms of using several OCL tools provided by the Toolkit. All models and metamodels are stored in a metadata repository. We use NetBeans' metadata repository implementation (hereafter referred to as MDR) as the Dresden OCL Toolkit also uses this facility. Meta Object Facility (MOF) is an OMG standard [Object Management Group, 2002] for defining metamodels and metadata repository, and NetBeans'

[1] http://www.cs.rmit.edu.au/agents/pdt

[2] The Dresden OCL Toolkit is an open source project providing various tools for OCL http://dresden-ocl.sourceforge.net

MDR[3] is one of its well-known implementations. Using the MDR and our metamodels, we are able to generate Java interfaces (JMI[4] compliant) for accessing models in the MDR.

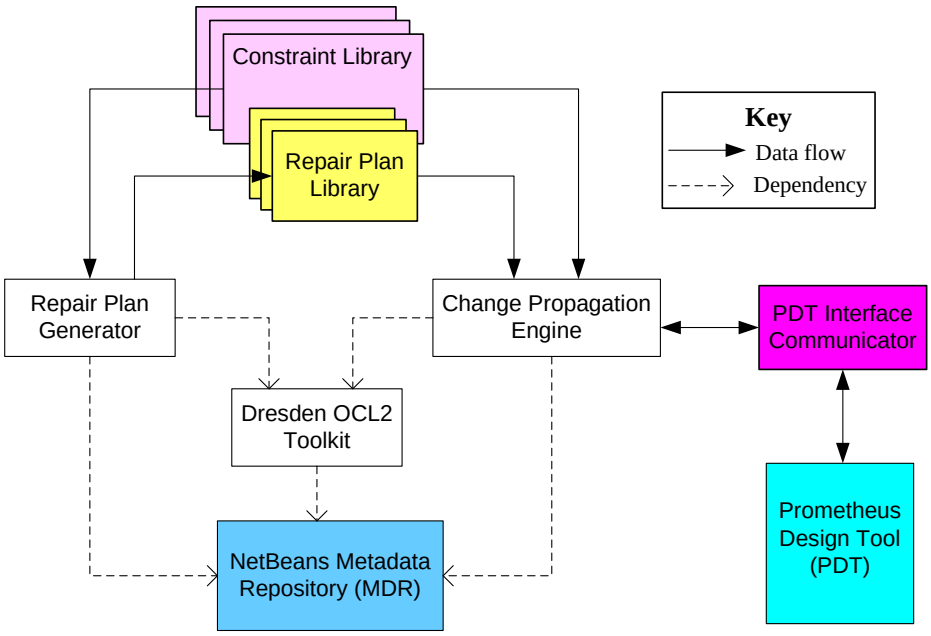


Figure 8.1: Change propagation assistant tool architecture

The Repair Plan Generator module is responsible for the repair plan generation process. This module parses input OCL constraints and a metamodel and creates a Constraint Library, which contains our own data structure of the input constraints. Each constraint in the Constraint Library has a Repair Plan Library, which stores plans for making the constraint true and false.

The Change Propagation Engine module is mainly responsible for propagating changes. It is integrated with the Prometheus Design Tool. We have developed a PDT Interface Communicator that provides an API for connecting the Change Propagation Engine and PDT. The PDT Interface Communicator also notifies the engine of user changes to the design model and the request for help from our engine in propagating changes. At the same time, the PDT Interface Communicator also sends the current (Prometheus) design model to the engine, which converts it to a MOF-compliant model stored in a NetBean MDR repository.

The main processes in the Change Propagation Engine include identifying which con-

[3]http://mdr.netbeans.org
[4]The Java Metadata Interface (JMI) standard is available at http://java.sun.com/products/jmi.

straints in the Constraint Library are violated and if there are any, finding plans in the Repair Plan Library for fixing them. The engine also performs plan selection which involves look-ahead cost calculation, and presents a set of best repair options to the user. If the user accepts one of the options proposed, it then instructs the PDT Interface Communicator to apply those changes to the current Prometheus design model in PDT.

8.2 Dresden OCL2 Toolkit

The Dresden OCL2 Toolkit[5] provides a platform with various forms of OCL tool support. The Toolkit is an open source project, designed to support openness and modularity. One of its major objectives is to allow other developers to integrate and adapt the existing OCL tools (provided with the Toolkit) into their own environments. The Dresden OCL2 Toolkit supports OCL 2.0 and UML 1.5, and consists of four major parts: the Metadata Repository, base tools, tools working on the base tools, and end user tools (refer to figure 8.2).

- The Metadata Repository (MDR): all tools provided by the Toolkit are metamodel-based and depend on a common metamodel derived from the MOF 1.4 and UML 1.5 metamodels. The Toolkit uses NetBeans' MDR to store all models and metamodels, including MOF 1.4, UML 1.5 and a common OCL metamodel. When models are loaded into the MDR, Java APIs (JMI compliant) for accessing metadata described by the specified MOF metamodel are automatically generated by the NetBeans' MDR infrastructure. The whole Toolkit uses these Java interfaces to access models stored in the MDR. It is noted that NetBeans' MDR is able to load only MOF 1.4 metamodels in the form of an XMI document. Therefore, in order to use the Toolkit our models and metamodels have to be compliant to MOF 1.4.

- The base tools of the Toolkit consist of three different modules. First, the OCL2Parser transforms the input OCL 2.0 constraints into an abstract syntax tree. The abstract syntax tree plays a key role in the whole Toolkit since it forms the common data representation for all other tools in the Toolkit. The OCL2Parser uses the popular LALR(1) parser generator SableCC[6] to build lexer, syntax analyser and an abstract attribute evaluator skeleton. Second, the OCL Base Library provides an implementation of the

[5] http://dresden-ocl.sourceforge.net
[6] http://www.sablecc.org

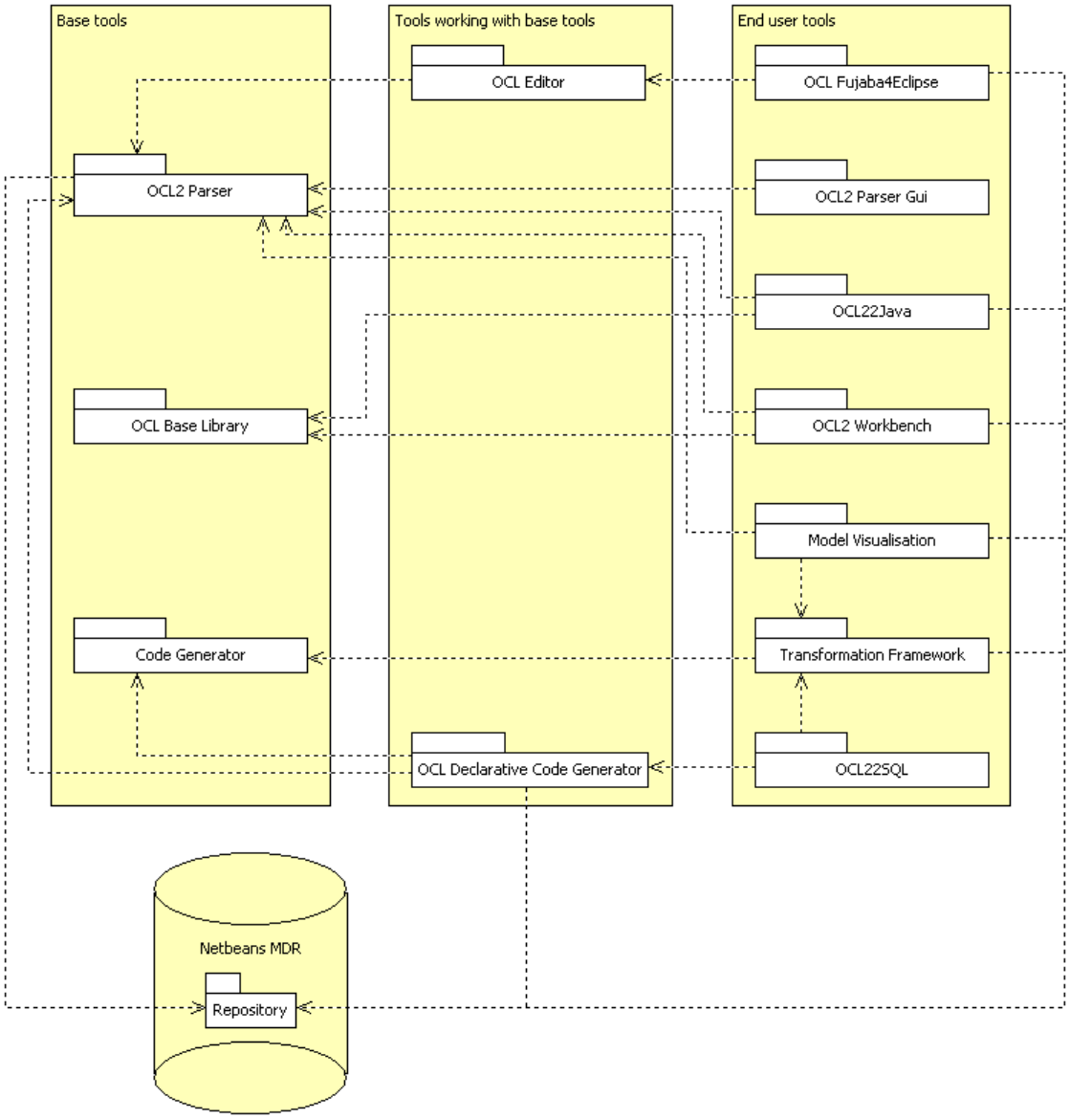


Figure 8.2: Packages and Tools of the Dresden OCL2 Toolkit (copied from http://dresden-ocl.sourceforge.net)

OCL Standard Library including OCL types, constructs, and expressions. Third, the Code Generator generates Java code for a given OCL expression (which can be either a query or instrumented Java code that evaluates the OCL constraint).

- Tools working with base tools include an OCL Editor, which is a text editor for OCL constraints, and OCL Declarative Code Generator which uses the Code Generator component to generate declarative target code (e.g. Java or SQL) for a given expression in OCL.

- End user tools are a series of stand-alone tools developed for a wide range of usage scenarios, using the base Toolkit infrastructure. For example, OCL Fujaba4Eclipse is an integration of parts of the Toolkit to Fujaba4Eclipse[7], an integrated teaching environment based on Eclipse. Other examples of end user tools include OCL22SQL, which generates SQL code from a given UML 1.5 model and OCL invariants, or OCL22Java, which is able to generate Java code from OCL expressions and instruments Java programs with the generated code. More details of end user tools and other tools provided with the Toolkit are available at the project home page.

Our Change Propagation Assistant uses the base tools of the Dresden OCL Toolkit. Details of how such tools are used in our implementation are discussed in the next sections.

8.3 Repair Plan Generator module

The plan generator module consists of three major packages. Figure 8.3 depicts how these packages are connected to each other. Each of these packages is discussed below.

Constraint Processor package

The Constraint Processor uses the OCL2Parser provided by the Dresden OCL2 Toolkit to process input OCL constraints and their associated metamodel in the form of an XMI document. The OCL2Parser transforms the input OCL constraint into an abstract syntax tree and a SableCC tree walker. Our Constraint Processor customizes this tree walker to build up constraint objects, our own data structure that represents OCL constraints. Since the current OCL2Parser only supports UML 1.5 models, only metamodels that conform to the

[7] http://wwwcs.uni-paderborn.de/cs/fujaba/projects/eclipse/index.html

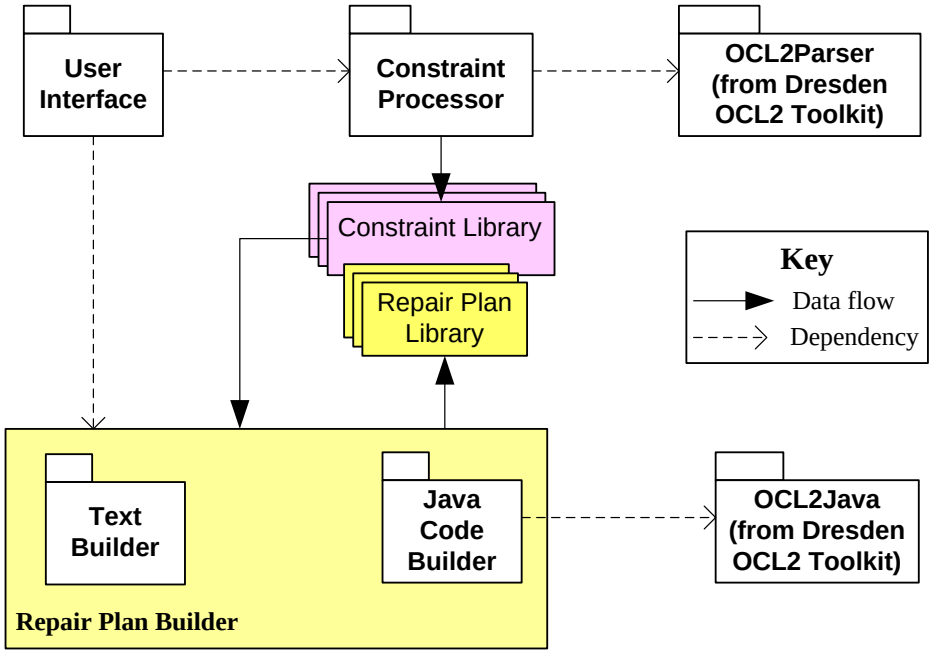


Figure 8.3: Packages of the Repair Plan Generator module

UML 1.5 metamodel are accepted by the Constraint Processor. For example, the Prometheus metamodel has been developed and serialized as an XMI representation of a UML 1.5 model.

In addition, we have also added several important features to the Constraint Processor. Firstly, it automatically creates constraints corresponding to the cardinalities of associations expressed in the metamodel. All constraints created by the Constraint Processor are stored in the Constraint Library.

Another feature of the Constraint Processor is the ability to generate a MOF 1.4 compliant model, which is equivalent to the input metamodel (UML 1.5 compliant). The need to have a MOF 1.4 compliant metamodel is due to the fact that the NetBeans' MDR only accepts a MOF-compliant metamodel as input. As a result, we use the UML2MOF[8] tool to convert the input metamodel into a MOF-compliant metamodel.

Repair Plan Builder packages

The Repair Plan Builder uses the repair plan translation scheme defined in chapter 6 to create repair plans for constraints stored in the Constraint Library. We support two types

[8] http://mdr.netbeans.org/uml2mof

of repair plan builder: the Text Builder that generates textual representation of repair plans (in the form of the AgentSpeak-based abstract syntax that we have defined), and the Java Code Builder that generates repair plans in Java code. Both of the repair plan builders are implemented using the Visitor pattern [Gamma et al., 1995] in which they contain visitor methods which traverse the structure of a constraint object and build up several sets of repair plans for each of the constraints (including one for making the constraint true, one for making the constraint false, and the others for addition and deletion involving derived sets if necessary). The Java Class Builder also uses the OCL22Java component of the Dresden OCL2 Toolkit to generate fragments of Java code that perform constraint evaluation. Repair plans generated from the Repair Plan Builder are stored in the Repair Plan Library.

User Interface package

Through the User Interface package, the user is able to provide inputs to the Repair Plan Generator, and view and modify generated repair plans. Figure 8.4 shows a screenshot of the user interface. The user (i.e. repair administrator) is required to provide an XMI file containing a metamodel and a text file containing a set of OCL constraints[9]. The user can also decide whether cardinality constraints in the input metamodel should be generated and stored in the constraint library by checking the "Generate cardinality constraints from the metamodel" option. There is also another checkbox that allows the user to choose whether a MOF 1.4 compliant metamodel equivalent to the input metamodel is generated.

The user interface of the Plan Generator also allows the user to view and modify the generated repair plans. Figure 8.5 shows a screenshot of the repair plan editor pane. On the left hand side is a list of constraints. The constraints are arranged in a hierarchical structure. It displays the context of a constraint which is a model entity. Below each context is a list of constraints associated with the context. For each constraint, it displays the sub-constraints and any other events posted within the top-level repair plans.

The right hand side contains two panes. The top pane displays the OCL expression of a particular constraint that is currently selected on the left-hand side tree. The bottom pane displays a textual representation of the repair plans of the selected constraint. There are two types of repair plans: for making the constraint true, and for making it false. The user can view each of them by switching between two tabs "Making-true plans" and "Making-false

[9]The user can create and edit OCL constraints using any available OCL Editor.

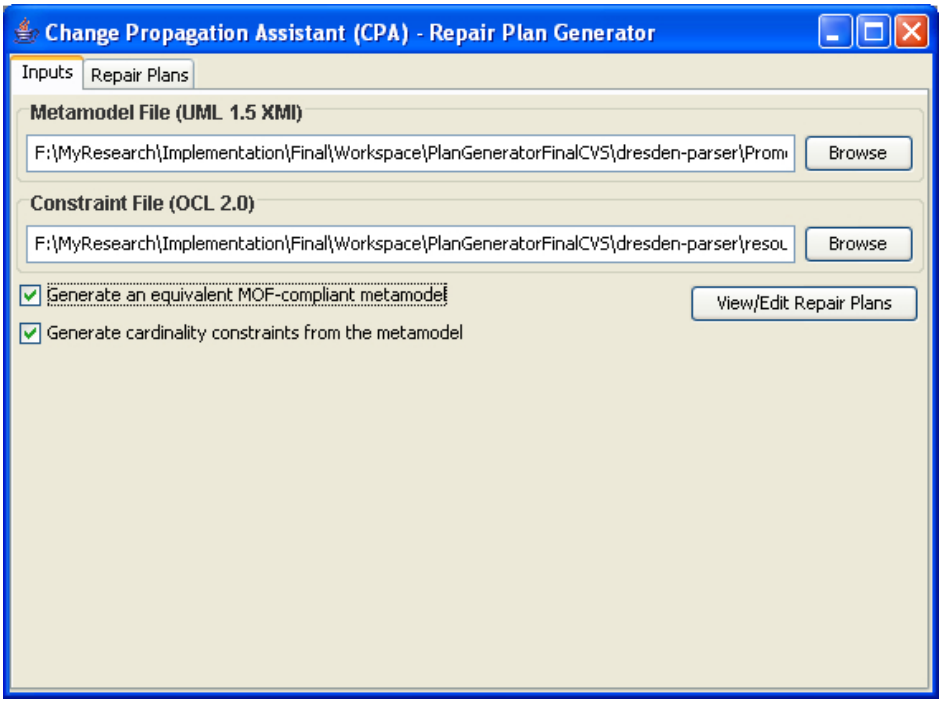


Figure 8.4: Screenshot of Repair Plan Generator's inputs pane

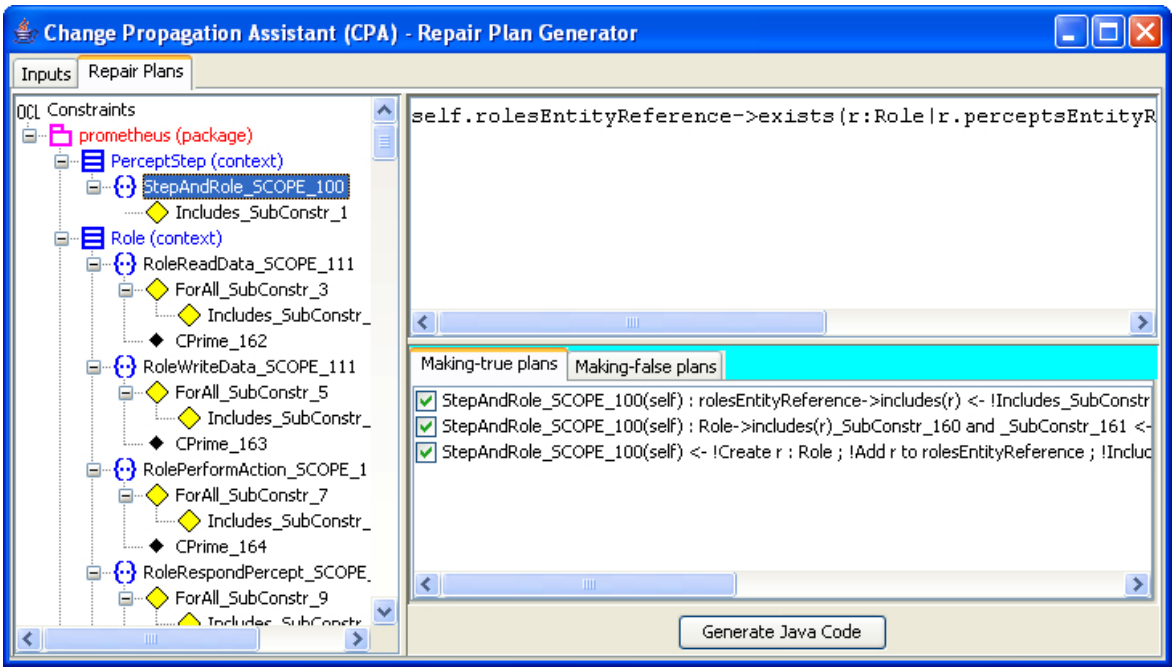


Figure 8.5: Screenshot of Repair Plan Generator's repair plan editor pane

plans". Relevant plans for addition and deletion involving derived sets appear in both of the two views. The user is allowed to modify the repair plans in terms of enabling or disabling a particular repair plan by selecting or deselecting the checkbox next to each repair plan. An advanced version of the tool will also allow the user to modify the context conditions of repair plans, e.g. to include domain specific conditions. However, when modifying repair plans, the user needs to be careful not to compromise the completeness of repair plans.

Once the user finishes viewing and editing, he/she can click on the "Generate Java Code" button to trigger the Java Code Builder to generate Java classes forming the Repair Plan Library.

8.4 Change Propagation Engine module

The Change Propagation Engine is responsible for checking constraints and finding repair plans for resolving constraint violations. The engine works on a "live" design model which is currently modified by the designer. As a result, the engine needs to be integrated with a modelling tool. Figure 8.6 shows how the engine is integrated with the Prometheus Design Tool (PDT). The Change Propagation Engine contains two major components, Constraint Evaluator and Cost Calculator, which are discussed below.

Constraint Evaluator

This package is responsible for checking constraints stored in the Constraint Library. It begins with creating constraint instances by binding each constraint type with its corresponding context instances (i.e. model entity instances) in the model. For each constraint instance, the Constraint Evaluator executes code (generated by Dresden Toolkit's OCL2Java earlier) that checks whether the constraint is violated or not with respect to the given model. Violated constraints are presented to the user through the User Interface component of the PDT Interface Communicator.

Cost Calculator

This component uses the "live" design model stored in the MDR and the repair plan instances stored in the Repair Plan library to calculate repair plan costs. The component implements the cost algorithms discussed in chapter 7. Cheapest repair plans are returned and presented to the user through the User Interface component of the PDT Interface Communicator.

The PDT Interface Communicator provides a connection between the Change Propagation Engine and PDT. It contains the following components.

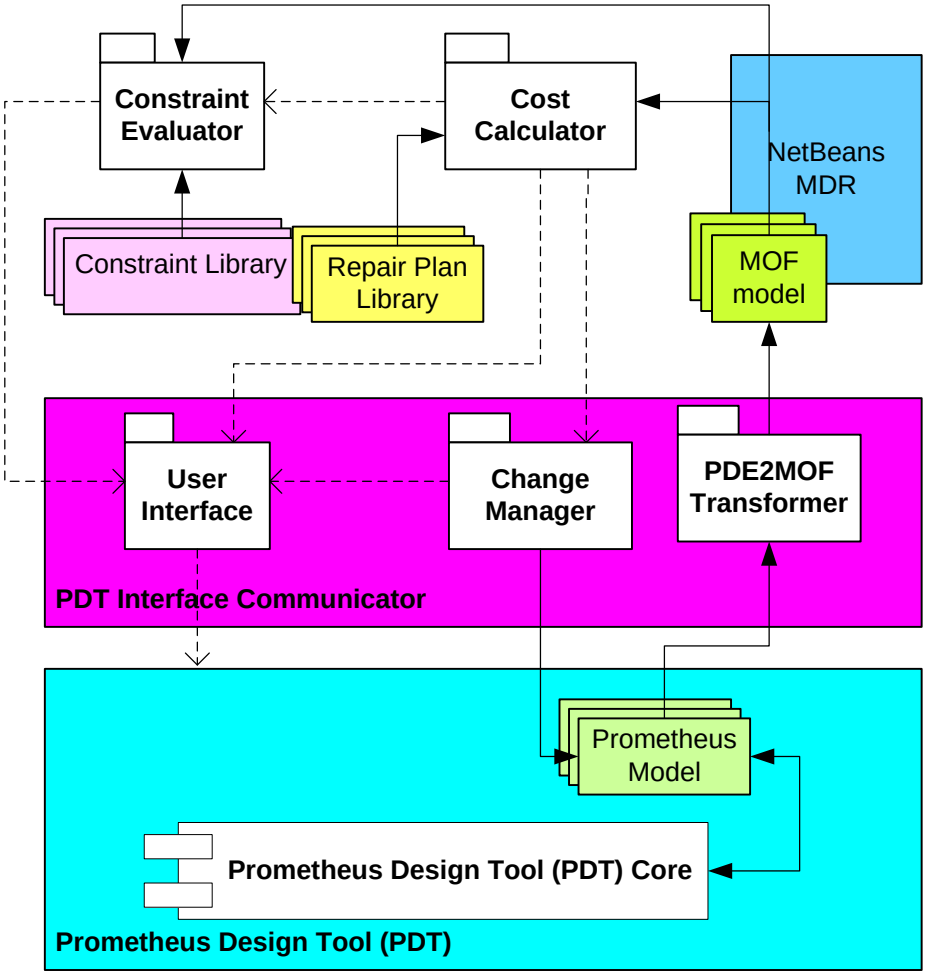


Figure 8.6: Packages of Change Propagation Engine and PDT Interface Communicator

PDE2MOF Transformer: Since our change propagation engine relies on models stored in MDR, it is necessary to convert input models to MOF-compliant models. PDE2MOF Transformer takes the current Prometheus (design) model and converts it to a MOF 1.4 compliant model (to be stored in the MDR). PDT is able to create a design model in terms of an XML document (known as a PDE). As MOF models are also represented using XML, we used XSLT[10] to implement the PDE2MOF Transformer.

User Interface: This component is plugged into the user interface of PDT to allow the

[10]http://www.w3.org/TR/xslt.html

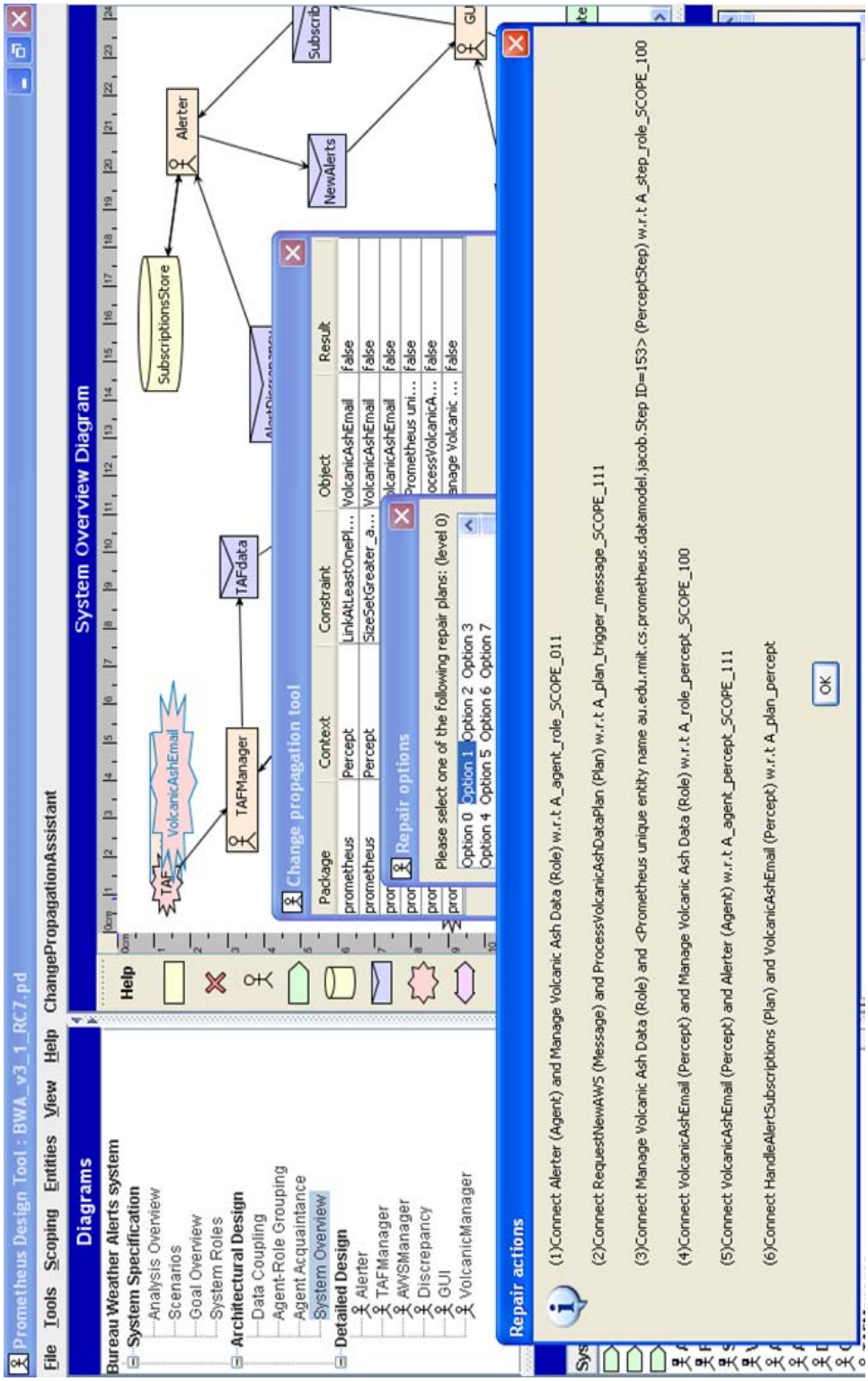


Figure 8.7: Screenshot of change propagation user interface in PDT

designer to interact with the change propagation engine. Firstly, through the user interface the designer is able to invoke the change propagation engine when he/she wants to. Secondly, violated constraints are presented to the designer through the user interface (as shown in the "Change propagation tool" popup window in figure 8.7) . The designer is also able to decide which violated constraints should be fixed or ignored by changing the evaluation result of a constraint from false to true. Thirdly, the user interface is also responsible for presenting cheapest repair plans and their details that are identified by the engine's cost calculator. Finally, through the user interface, the user is also able to configure the basic cost settings, i.e. assign costs to primitive repair actions. Figure 8.7 shows a snapshot of the interaction between the user and the change propagation engine through the user interface of PDT.

Change Manager: Change Manager is responsible for executing repair plans (chosen by the designer through the user interface) onto the actual design model in PDT. In order to do so, it has a mapping between primitive actions in our repair plans and the actual change actions allowed in PDT. Change Manager also records primary changes made to the model, and gives them to the Cost Calculator. In case there is a large number of cheapest repair plans found by the Cost Calculator, primary changes are then used as input to a heuristic that aims to make it faster for the user, by placing plans that are more likely to be chosen earlier in the list. More specifically, new model entities that have been created by primary changes are identified. The Cost Calculator then sorts the returned (cheapest) repair plans based on the number of their contained repair actions that affect those new entities. Repair plans that have more actions involving the new entities appear earlier in the list presented to the user. This particular heuristic is based on the assumption that "desirable" secondary changes are likely made to the newly created entities so that they form consistency relationships with other existing entities in the model. This heuristic has not, however, been thoroughly developed or evaluated, which is a topic for future work.

8.5 Chapter summary

In this chapter, we have introduced a prototype tool support for change propagation, our Change Propagation Assistant tool, that is integrated with the Prometheus Design Tool (PDT). We have presented the architecture of the tool, described its major components, and explained how it is integrated into PDT. Since this tool is a proof-of-concept of the ideas proposed in this dissertation, no experiments with end-users can be done because

of the primitive user interface of the tool. We recognise that there is a range of possible improvements and extensions that can be applied to our tool.

However, we were able to show how our change propagation framework, as introduced in chapter 3 and discussed in more detail in chapters 6 and 7, can be implemented and realised by tool support. In addition, the implementation architecture indicates that Change Propagation Assistant can not only be integrated with PDT, but also can be modified to be plugged into other modelling environments. The tool is also used to evaluate the effectiveness and efficiency of our approach, which is the topic of the next chapter.

Chapter 9

Evaluation

Having implemented our framework for a change propagation assistant, we would now like to perform an empirical evaluation of the approach and tool. The focus of our evaluation is to measure two critical aspects of our approach: effectiveness and efficiency. We first describe our methods of evaluation and discuss the issues posed by the evaluation. We also introduce a taxonomy of changes, present a change propagation process based on the designer's behaviour, and describe the evaluation process and metrics. We then discuss and analyse the results of an evaluation application. Finally, we assess the efficiency of our approach and also examine its scalability by looking at the application of our approach in artificial settings.

9.1 Issues in evaluation

Our evaluation needs to address the effectiveness and efficiency of our approach. This involves answering several key questions. Firstly, how well this approach works in practice and, specifically, how useful is it likely to be to a practising software designer who is maintaining and evolving a system? In addition, how well does it scale to larger problems? Unfortunately, an evaluation to answer these questions raises a number of challenges and further questions. In this section we discuss the nature of these issues, and present and justify our solutions to each of them.

Ideally, the evaluation would be done by giving the Change Propagation Assistant (CPA) tool to a group of selected users, who would be asked to work with the tool to implement requirement changes. We would then collect and analyse data such as their feedback, the change outcomes, etc. to assess how much assistance the CPA tool provides them. However,

due to time and resource limits, we were not able to conduct such a comprehensive user study evaluation.

In order to overcome this obstacle, we approached the evaluation by defining an abstract user behaviour in maintaining/evolving an existing design. More specifically, we defined a process of change propagation based on the user's behaviour. The key item of this process is a user's change plan. We then simulated a real user by performing steps in this plan and assessed the responses from the CPA tool. Details of this methodology will be discussed in section 9.3.

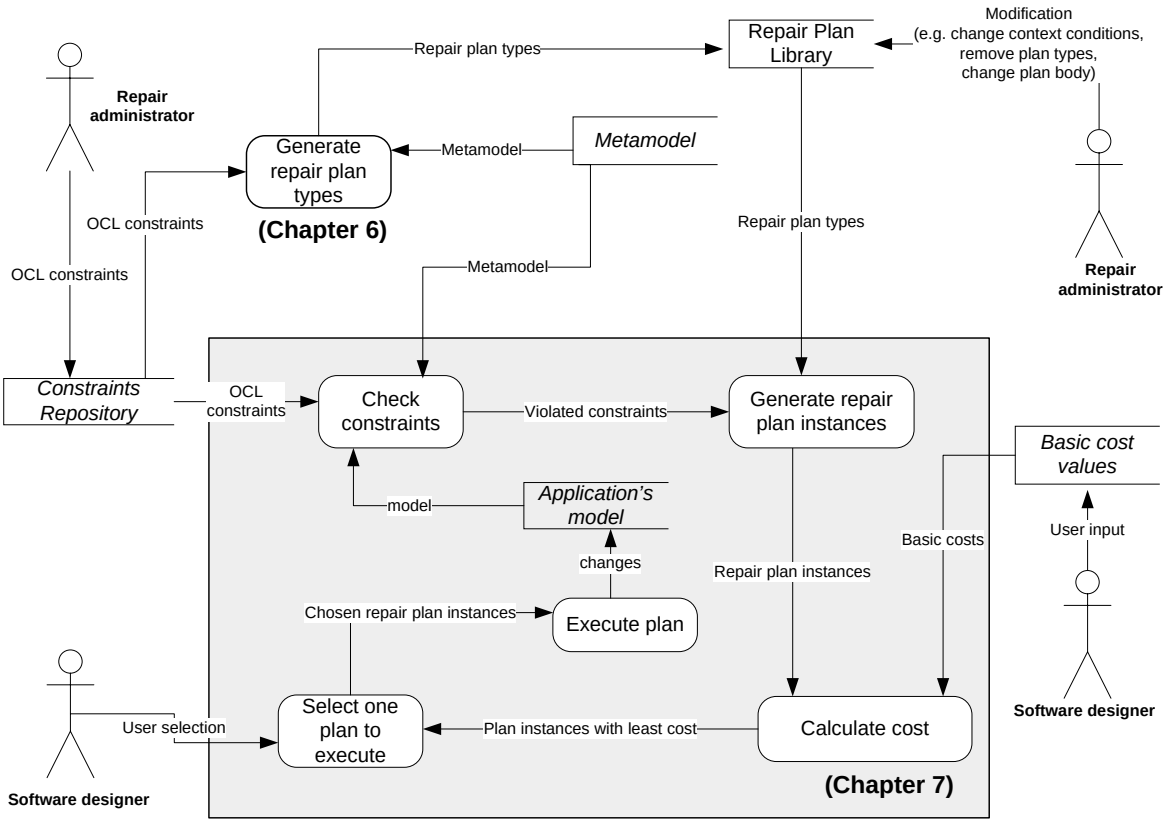


Figure 9.1: Change propagation framework

We now consider the specific issues that are raised by an evaluation (whether it involves users or not). These issues relate to the choice of inputs to our change propagation framework and the CPA tool (see figure 9.1) including: which methodology? which applications? which requirement changes? which primary changes? and what basic costs?.

9.1.1 Which methodology should be used?

As discussed earlier, our approach is applicable to various design methodologies. However, time and resource constraints do not allow us to investigate a large range of methodologies. Rather, it is more feasible to select one or two prominent methodologies and apply our approach to them. The candidate methodology, however, needs to satisfy several criteria. It is important that the methodology's metamodel and consistency constraints are available or that they do not require substantial time and effort to develop. As a set of constraints is an important input to our change propagation framework, it is also necessary to justify the choice of constraints, e.g. using existing constraints or developing them based on the methodology's documentation. In addition, there are other issues that we should also take into account: ideally, we want a mature and complete methodology that has been widely used in academia and industry.

Given the popularity of UML in both academia and industry, an application of our approach to UML[1] would be ideal. In fact, our original plan was to use UML design models to assess our approach, but the effort involved in implementing all of the constraints in the UML standards was beyond our resources. Furthermore, we want to focus on assessing how our approach helps improve support for software evolution in agent systems. As a result, we have chosen to use the Prometheus methodology [Padgham and Winikoff, 2004] for the design of agent systems as our key evaluation subject. Prometheus is a prominent agent-oriented software engineering methodology that has been used in both academia and industry. The Prometheus notation is simpler than UML, and in addition to local expertise, we had easy access to the source code of the Prometheus Design Tool (PDT), allowing the Change Propagation Assistant to be integrated with PDT. We defined a Prometheus metamodel and a set of consistency constraints (see chapter 4) by examining the well-formedness conditions of Prometheus models, the coherence requirements between Prometheus diagrams, and the best practices proposed in [Padgham and Winikoff, 2004].

9.1.2 Which application(s) should be used?

The main input to the change propagation framework is the application's existing design. Therefore, in order to maintain the credibility of our evaluation, the choice of application needs to meet several requirements. Firstly, it is important for the application to have a set of

[1] UML here is in fact a shorthand for something like "UML plus a suitable design process".

justified, realistic, maintenance/evolution changes. Ideally, changes should be motivated by actual change requirements of real applications. This makes the evaluation more convincing since it avoids the possibility of bias by using existing change requirement. Secondly, the application must have been developed using the chosen methodology. Thirdly, the application's existing design should either be available or be easily reverse engineered from the software system. Furthermore, the application needs to be of a reasonable size: not too small (which affects the credibility of the evaluation) but not too large (which makes the effectiveness evaluation harder to perform). Finally, applications originating from industry would have a higher preference because there is no need to justify that they are real applications.

However, it was difficult to access existing applications that satisfy all the above criteria. Industrial applications do not usually have disclosed designs whilst open-source projects often have limited design documentation and would require a substantial effort to reverse-engineer. In addition, in the case of an agent-oriented methodology, there are few available agent applications with designs.

As Prometheus is the selected methodology, the choice of application is within systems that have been developed using the Prometheus methodology. We fortunately had access to the design of an agent-based weather alerting system, which satisfies many of our criteria. The original weather alerting system was developed between 2002 and 2005, as part of a collaboration between the Australian Bureau of Meteorology[2], RMIT's Agents Group[3], and Agent-Oriented Software Pty Ltd[4]. This system was subject to various changes during its evolution [Mathieson et al., 2004]. Our evaluation is, to some extent, based on this application with such a source of real modifications. More specifically, we adopted simple versions of the system that have been used in [Jayatilleke, 2007] and in a student project in the agent-oriented programming course taught at RMIT University. Those versions of the system simplified the original application but captured its essence by retaining the key characteristics. In particular, the work in [Jayatilleke, 2007] specified several changes on a simple version of the original system, some of which were actual real changes and others were similar in type, suitable for the simplified system. We used this work as a major source to specify an initial

[2]The Australian Bureau of Meteorology is the National Meteorological Authority for Australia. Its role is "to observe and understand Australian weather and climate and provide meteorological, hydrological and oceanographic services in support of Australia's national needs and international obligations.". More details are available at http://www.bom.gov.au.

[3]http://www.cs.rmit.edu.au/agents

[4]http://www.agent-software.com

system with various types of changes.

9.1.3 What changes to the application should be done?

In order to maintain the subjectiveness of evaluation, it is important to select and justify the changes. This can be done by selecting changes that originate from well-motivated and realistic change scenarios in a real application. In addition, changes need to cover a range of different types. Even when the changes to be made originate in actual changes in a real application, if the changes do not cover the different types of changes, then the generality of the evaluation is weakened.

In order to ensure that we sufficiently cover different types of change, we developed a taxonomy of changes by adopting existing classifications. We then checked how well a proposed collection of changes covered the different types of change using the taxonomy. The focus of our change taxonomy resides in the application's design (i.e. the model) rather than other aspects of software maintenance and evolution (e.g. management). We also want to cover changes that reflect different purposes of maintenance. As a result, we adopt Swanson's classical taxonomy (described in chapter 2) as the primary, abstract, classification of change types. In addition, based on the classification proposed by Chapin et al. [2001] we introduce a second taxonomy where changes are categorised based on the types of modification made to the behaviour of a software system. More specifically, they can be: adding a new functionality, removing an existing functionality and modifying an existing functionality. For instance, the inclusion of volcanic ash warnings in a weather alerting system can be classified as perfective maintenance and functionality addition. By contrast, the removal of pressure alerts from the same system due to pressure data from the forecasters no longer being available can be regarded as adaptive maintenance and functionality removal.

Figure 9.2 depicts the change taxonomy that we used to classify requirement changes that were used to evaluate applications. As a check of coverage, we classify changes according to this taxonomy. It is, however, noted that in practice there is not always a clear-cut distinction between different types of maintenance. For instance, when adapting the software to a new environment (adaptive maintenance), functionality may be added to take advantage of new facilities supported by the environment (perfective maintenance) or removing a functionality may require modifying other existing functionalities that use the functionality to be deleted. In addition, we do not look at other types of changes such as different forms of refactoring

	Perfective	Adaptive	Corrective
Addition	Add a new functionality in response to business change	Add a new functionality to cope with an environmental change	Add a new functionality to fix bugs
Modification	Modify an existing functionality in response to business change	Modify an existing functionality to cope with an environmental change	Modify an existing functionality to fix bugs
Removal	Remove a functionality in response to business change	Remove a functionality to cope with an environmental change	Remove a functionality to fix bugs

Figure 9.2: A taxonomy of software change

[Fowler and Beck, 1999; Opdyke, 1992] in which a software system is changed in such a way that improves its internal structure but does not alter its external behaviour.

9.1.4 How do we select primary changes to perform?

The choice of primary changes is important because it affects the usefulness of the tool. If the primary change is too small, then the design may remain consistent and consequently there is little room for the tool to help. If the primary change is too large, then the tool again cannot help because the work has been done. We deal with the issue of selecting primary changes by not selecting: our evaluation process considers all of the possible primary changes.

In order to meet a new requirement, a given change to the system's design has to be made. We view such a change (denoted by D) as being a sequence of steps, with each step being a primitive change to the model such as adding or removing a link between entities. For each given change, D, we need to consider all possible primary changes P (with the actions in P being a subset of those in D). Now in fact considering arbitrary subsets of D is not meaningful as will be discussed in section 9.2: a designer will not select a random collection of actions from D as a primary change. Instead, they will perform an initial segment of D (i.e. P will be a prefix of D) and then invoke the CPA tool. Therefore, for each given change plan the possible primary changes to be considered are the possible initial segments of the change plan.

9.1.5 How are basic costs determined?

We need to select values for the basic costs of repair action. The costs assigned to basic repair actions can affect the change options proposed by the CPA tool. For instance, we

can discourage deletion by assigning a high cost to the removal action. As a result, the effectiveness of the tool may also depends on the basic costs. Therefore, it is important to consider the determination of these costs.

The designer can adjust the basic costs. In this evaluation we do not explore a range of costs, but instead select what we believe are reasonable values for each change scenario (explained with more detail in section 9.5). A thorough exploration of the effects of varying the basic costs is an interesting and important topic for future work.

9.2 A model of the change propagation process

As mentioned earlier, time and resource limits prevent us from conducting a real user validation of our approach in which we could understand how the user interacts with the CPA tool and consequently assess the usefulness of the tool. One step towards dealing with this issue is to model the change propagation process and thus capture the user's behaviour in changing the software. As discussed in section 2.1.4, several previous work have proposed to model the change propagation process (e.g. [Arnold and Bohner, 1996; Rajlich, 1997]). Based on the intuition and ideas proposed in these models, Hassan and Holt [2004] have recently proposed a simplified model of the change propagation process (see figure 9.3) and this is the one that we adopt.

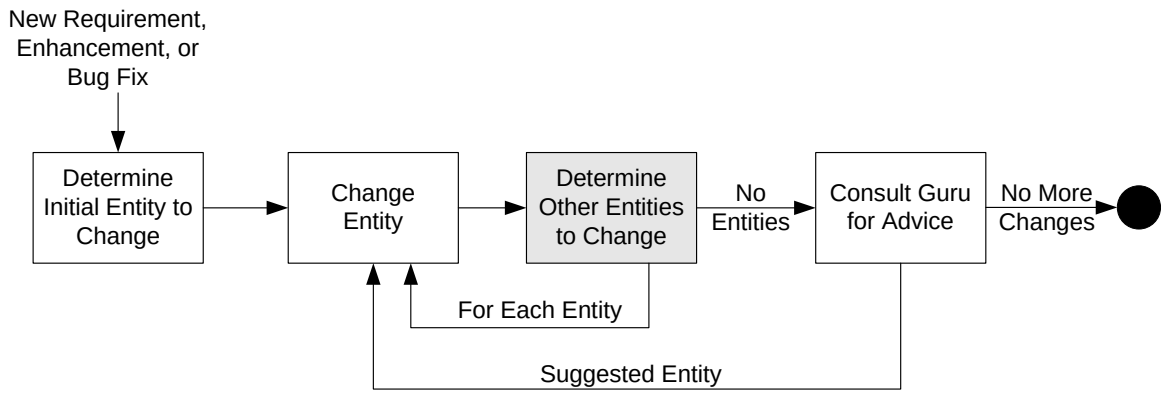


Figure 9.3: A model of the Change Propagation Process (redrawn from [Hassan and Holt, 2004])

In this model, the developer is guided by a change request to perform primary changes (i.e. determine initial entity to change and change entity) and some partial secondary changes (i.e. determine other entities to change). When the developer cannot locate other entities

to change, she/he consults a Guru. If the Guru indicates that an entity was missed, then it is changed and the change propagation process is repeated for that entity. This continues until all appropriate entities have been changed. The Guru can be a senior developer, a software development tool, or even a suite of tests. At the end of this process, the developer has determined the *change set* for the new requirement at hand and ideally all appropriate entities should have been updated to ensure consistent assumptions throughout the system.

We have used the intuitions and ideas proposed in the above change propagation model to formally define a process that captures (abstractly) the designer's anticipated behaviour and use this in our evaluation. Figure 9.4 shows our model of the change propagation process. In this model, the designer is guided by a change request which can be fulfilled by making a given change to the system's design. We view such a change plan (denoted by D) as being a sequence of steps, with each step being a primitive change to the model such as adding or removing a link between entities, which transforms the existing design model so that it meets the requirements of the change request. Although the designer does not have a complete knowledge of the change plan D, he/she may know some parts of it, especially an initial segment of D (denoted as P) where he/she determines initial entities and makes changes to them (i.e. performs a primary change). In addition, we also assume that the designer has a general idea of the change propagation direction and thus given a number of change options they are able to identify those which are compatible with the change plan. We assume for evaluation that a specific change plan D plan is given.

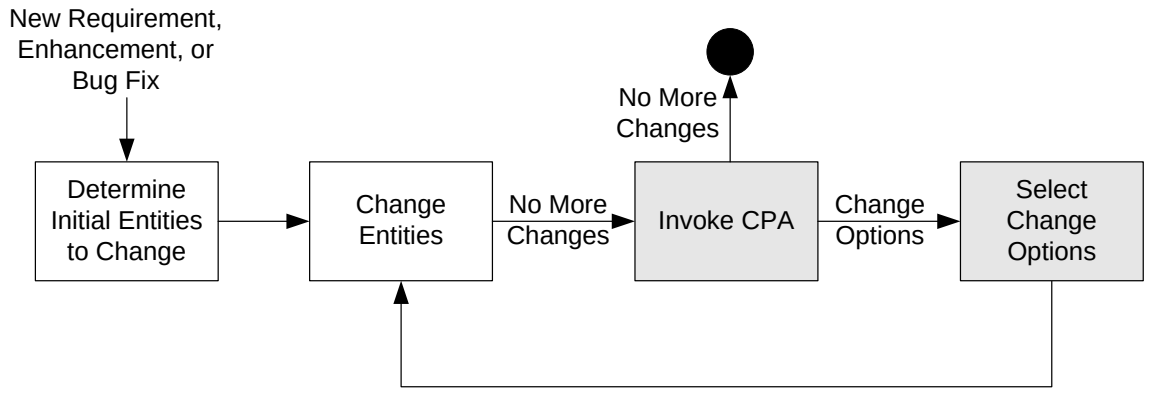


Figure 9.4: Our model of the Change Propagation Process

As can be seen, our CPA tool is considered as the Guru, but unlike the Guru in figure 9.3 it suggests not only the entities to be changed but also the specific changes to be made

1. given a change plan D (sequence of actions)
2. select $P \sqsubseteq D$ and do the actions in P
3. update D ($D := D - P$)
4. invoke the tool yielding $\mathcal{O} = \{C_1, \ldots, C_n\}$
5. if $\exists\, C_i \in \mathcal{O}$ where C_i is compatible with D then
6. select a compatible $C_i \in \mathcal{O}$ (if more than one)
7. do actions in C_i and update D ($D := D - C_i$)
8. end if
9. goto step 2 if D is not empty.

Figure 9.5: A change propagation process

to them. The designer performs some primary changes and then invokes the CPA tool. The tool then suggests a set of change options and the designer chooses one that is compatible with his/her plan. The tool then performs the actions in the selected repair option. After that the designer performs the next segment of his/her plan and this process continues until the change requirement is implemented.

Figure 9.5 formally describes our change propagation process in more detail. Given a change plan D, the designer selects an initial segment P of D and performs the actions in P (step 2), updates D by removing the performed actions (step 3) and then invokes the tool, which returns a set $\mathcal{O}$ of repair options (step 4). Each repair option C_i is a sequence of actions. At this point (step 5) the user may select one of the C_i (step 6) and apply it to the model (step 7), or they may decide that none of the C_i are suitable.

Deciding whether a C_i is suitable is done by comparing it with the designer's plan: C_i is *compatible* with D if all of the steps in C_i are in D (formally[5] $\forall\, s \in C_i \bullet s \in D$). If all the changes in D have been performed the process ends, otherwise the user continues to perform more primary changes (step 9). We use $D := D - P$ to denote removing the actions in P from the sequence D, and we use $P \sqsubseteq D$ to denote that P is an initial segment of D (formally $\exists\, X : P + X = D$, where $+$ is sequence concatenation).

[5] Alternatively, if viewed as sets, $C_i \subseteq D$.

We use the above interaction between the designer and the CPA to define our evaluation process and metrics which are described in detail in the next section. Although this process helps simulate the behaviour of a real user, it contains several limitations. The process is relatively abstract and does not cover other (potentially useful) details such as reasons for a design decision. In addition, the process is developed based on existing change propagation models [Arnold and Bohner, 1996; Hassan and Holt, 2004; Rajlich, 1997], but it is not clear to what extent these models have been validated by studying the practice of software maintenance and evolution by developers.

9.3 Experiment process and metrics

As pointed our earlier, efficiency and effectiveness are the two important criteria that we used to assess our approach in general and the CPA tool particularly. We dealt with the efficiency assessment by applying our approach in both real and artificial applications. In particular, we examined the scalability of our approach by simulating designs with variable numbers of model elements and constraints. With respect to the effectiveness evaluation, we used a "simulated user" who follows the change propagation process defined earlier in figure 9.5. More specifically, our evaluation process consists of the following steps:

1. For each new requirement we develop a change plan D. Because the tool's effectiveness is sensitive to the choice of change (i.e. D) it is very important to justify the change plan D with strong evidence showing that it is a reasonable design change. It is also noted that for a requirement there may be several alternative valid change plans that meet the requirement. In these cases, we should consider all of them. Furthermore, within the change plan D we mark "obvious" and "side-effect" actions separately. The differences between these two types of action are explained ahead.

2. We then apply the process described earlier (see figure 9.5), considering a partial change P that expands by one step at a time. By doing this, we consider all possible choices of primary change.

When the process terminates, we measure what proportion of the actions in the change was done by the user, and what proportion was done by the CPA. More specifically, we count how many actions ended up being in C_is along the way, compared with the total number of

actions in D so we calculate[6] the metric $M =\mid C \mid / \mid D \mid$ (where C is the union of the C_is). It is noted that the value of M depends on our choices for P. Clearly, if P is the whole of D then there is nothing left for the tool to do, and M will be 0. In some of the cases below we will see that there is a "tipping point": until enough of D is done the tool cannot help, but once enough is done, the tool performs the remaining steps in D.

In order to further assess the usefulness of our CPA tool, we also distinguish "obvious" change actions from "side-effect" actions. Obvious actions are changes that are part of the main design flow and/or in the scope that the designer is currently working in. For instance, if the designer creates a new agent, then his/her next steps would be developing the agent's internals, e.g. creating plans. In contrast, it is more difficult to identify "side-effect" actions, changes that are needed in other parts of the system that are neither part of the main design flow nor in the scope of the designer. As a result, the designer tends to miss those secondary side-effect changes. In the previous example, side-effects could be assigning the newly created agent with a role. It is clear that the tool is more useful if it can pick out such side-effect actions. As a result, we consider not only the percentage of actions in the change done by the CPA, but also their quality in terms of how many of them are side-effect actions. Therefore, we classify the repair actions into two categories: "obvious" and "side-effect" change actions and count them separately. In addition, since a repair option C_i returned by the CPA can contain "side-effect" actions, we need, for evaluation purposes, to require that the change plan D also contains all of the consequent side effects. In this sense, D is not really the designer's plan per se, but rather than the designer's plan plus consequent side effects.

In addition to measuring M, an important factor in the usefulness of the tool concerns the repair options, $\mathcal{O}$. Specifically, we are interested in how many of the C_i in $\mathcal{O}$ are compatible with D, and in the size of $\mathcal{O}$ (since it is clearly better if the designer is not being asked to select an option from a very large list). We thus, in addition to M, also measure the number of options and how many of the options are compatible with D.

9.4 An overview of the evaluation application

Our choice of application was the Bureau of Meteorology's multi-agent system for weather alerting (MAS-WA). The details, including the design and functions, can be found in [Mathieson et al., 2004]. The original system had been developed between 2002 and 2005, as part

[6]Note that, if viewed as sets, $D = C_i \cup P$.

of a collaboration between the Australian Bureau of Meteorology, RMIT Agent Group[7], and Agent-Oriented Software[8]. As discussed previously, we built an evaluation application based largely on simplified versions of this system which had been used for previous work in the group and for teaching.

The aim of the original MAS-WA application is to enhance aviation forecasts with a special focus on providing assistance in the rapid rectification of forecasts for airports and pilots [Dance and Gorman, 2002; Mathieson et al., 2004]. The system subscribes to various weather data sources such as terminal aerodrome forecasts (TAFs[9]) generated for airports, automated weather station (AWS) readings, email notification on volcanic ash (e.g. from the Volcanic Ash Advisory Center (VAAC)) and thunderstorm alerts. The system uses data from those sources to extract important information and send alerts (or warnings) to interested clients regarding situations such as extreme weather, inconsistencies between data sources, or changes to observed weather that contradict previously issued forecasts. The initial version of the system, which was developed using the JACK agent language [Busetta et al., 2000], had undergone different changes to extend and further develop the system. For instance, the alerts on volcanic ash levels and thunderstorms were introduced later.

In the rest of the thesis (and in chapter 4, pages 69-76 and 102-105), our description of the weather alerting system is the simplified application which we have used in our work. The weather alerting system monitors data from forecasts for airport areas (TAF) and from automated weather stations (AWS). TAF and AWS readings contain information about temperature, wind speed and pressure (see figure 9.6). In the (simplified) system alerts are issued if there are significant differences between a prediction (TAF) and the actual weather (AWS). There is a fixed discrepancy threshold for each data type for generating alerts for all alerting regions. Alerts are provided to human operators (e.g. airport officials) via a GUI, which subscribes to certain kind of alerts for particular airports, e.g. a subscription for temperatures alert at Melbourne airport. It is also important that a GUI is not overwhelmed with messages. As a result, a single airport should receive a single alert that contains multiple messages. In addition, if a warning has been sent to a GUI in the last 30 minutes, it should not be sent again if the situation is not changed.

The initial design of the system consists of five agent types: “Alerter”, “TAFManager”,

[7] http://www.cs.rmit.edu.au/agents

[8] http://www.agent-software.com.au

[9] TAFs stand for terminal aerodrome forecasts, highly abbreviated forecasts of weather around airports intended for pilots [Mathieson et al., 2004].

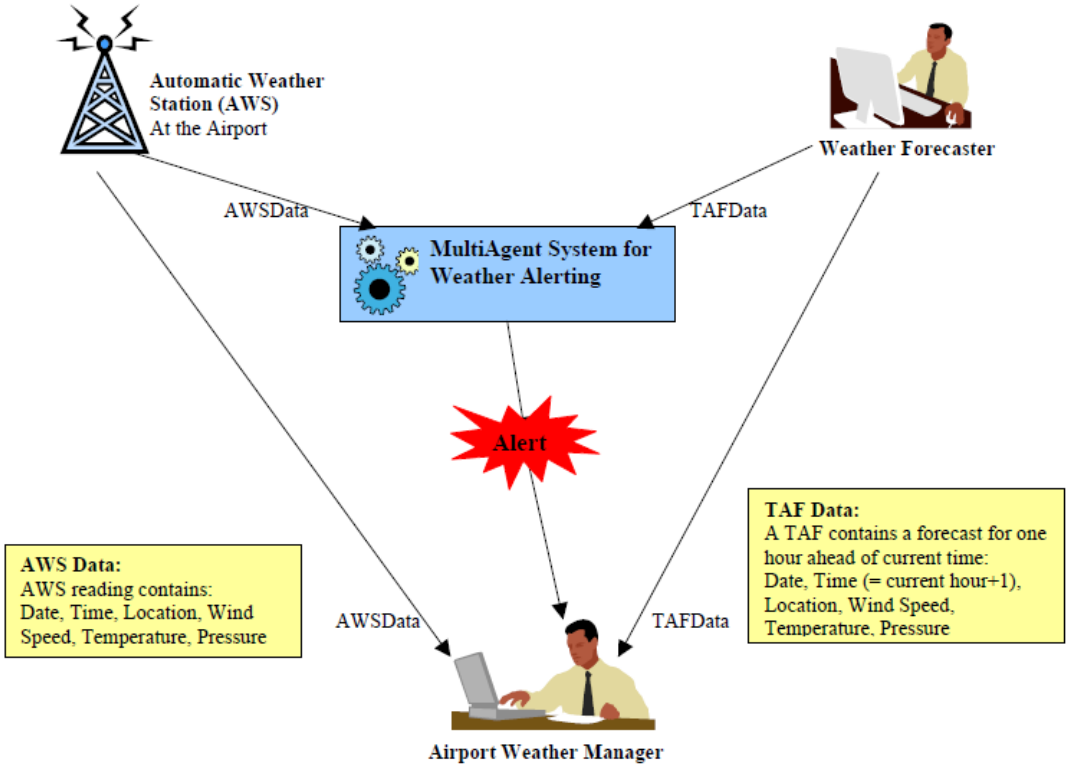


Figure 9.6: MAS-WA System Overview Diagram (from [Jayatilleke, 2007])

AWSManager", "Discrepancy" and "GUI". Figure 9.7 shows the system overview diagram for the application, as well as agent overview diagrams for the "Discrepancy" and "GUI" agent types. The "TAFManager" processes the "TAF" percept that contains "TAF" data, which is stored in the "TAFDataStore". Similarly, the "AWSManager" is responsible for extracting the "AWS" data in the incoming "AWS" percept and store this data in the "AWSDataStore". The "AWSManager" agent also notifies the "Discrepancy" agent by sending an "AWSData" message whenever new AWS is obtained. In addition, when the "Discrepancy" agent asks for new AWS (by sending a "RequestNewAWS" message) the "AWSManager" perform a "NewAWSRequest" action to request an external weather station for this information. The "Discrepancy" agent detects discrepancies between TAF and AWS, and sends an "AlertDiscrepancy" message to the "Alerter" agent if a discrepancy above the threshold level occurs. The "Alerter" agent then processes the alerts and sends to subscribed "GUI" agents.

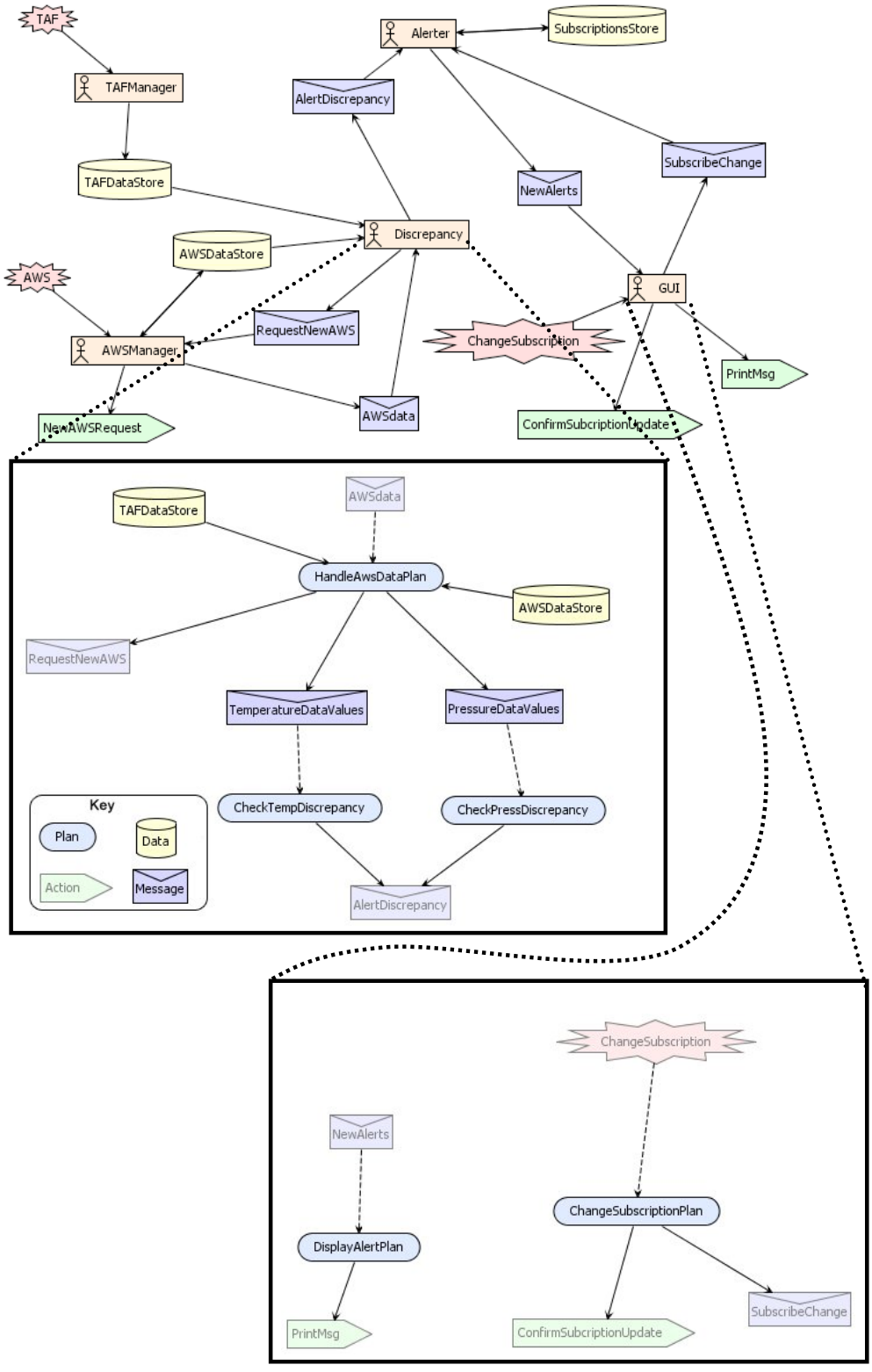


Figure 9.7: MAS-WA Design

9.5 Change scenarios and results

In this section, we present a set of changes that have been made to the MAS-WA application. Some of the changes were based on real changes but others were introduced by Jayatilleke [2007] and ourselves, reflecting typical types of changes needed. In other words, the changes were similar in type to actual changes that happened in the real application, but were substantially simplified to match the simplified system that we used. The changes to be made, however, do not cover all categories in the classification of changes discussed in section 9.1.3. Figure 9.8 shows how those changes are mapped against our change taxonomy. Of the 6 changes, 3 of them are perfective maintenance, 2 are adaptive maintenance and 1 for corrective maintenance. In terms of the functionality dimension, 4 of the 6 changes are adding new functionalities while 2 of them are related to functionality modification. We could not find any real change examples involving removing functionalities as functionality removal rarely happens in practice [Chapin et al., 2001].

	Perfective	**Adaptive**	**Corrective**
Addition	Change 4, 6	Change 3	Change 1
Modification	Change 2	Change 5	
Removal			

Figure 9.8: Changes are classified based on our taxonomy

We followed the evaluation process defined in section 9.3 to assess the effectiveness of our approach in each of the change scenarios described earlier. In terms of setting basic costs[10], for all six changes connection (between two entities) is assigned a cost of 1 and so is modification. Most of the changes are perfective and/or adding new functionalities, which tends to result in new entities in the design model. As a result, we set the cost of creation be 0 to encourage new entities to be created. In Prometheus, a creation of an entity, however, usually takes place within another entity, for example creating a plan within an agent. For such cases, we treat them as a single creation which also has a cost of 0. Disconnection and deletion are assigned a slightly higher cost (of 3) for most of the changes. The special case is change 5 which involves disconnecting entities and we set the cost of disconnection to be equal to the cost of connection (i.e. 1) for this case. Although the assignments of basic

[10]As mentioned in chapter 3 and 8, our framework and its prototype implementation (i.e. the Change Propagation Tool) allow the designer to change the basic cost configuration as a means of giving his/her

costs in these cases may be relatively arbitrary and example-specific, in our view they seem reasonable since they reflect the general user preference to create and link entities, rather than delete them. However, as discussed in chapter 10 more work is needed to explore the effects of basic cost selection and to determine to what extent generally good basic costs can be determined.

For each change we describe the change requirements, justify a change plan that meets the change requirement, and present the result of executing that change plan using our CPA tool.

9.5.1 Change 1: Adding wind speed alerting

The initial system only took care of discrepancies between weather readings (AWS) and forecasts (TAF) for temperature and pressure data. The first requirement change is to add discrepancy alerting for wind speed, given that relevant data values were available in the AWS and TAF. This, therefore, involves performing the necessary modifications to process wind data and generate alerts on any wind speed based discrepancies. We are viewing this change as aiming to correct an error in the design (inconsistent with the requirements[11]) by adding a new functionality. It is, therefore, classified into the category of corrective and addition of our change taxonomy.

As the "Discrepancy" agent is responsible for detecting data discrepancies, it needs to be changed to meet this change request. Similar to temperature and pressure, we need a plan for checking wind speed discrepancies. This plan is triggered by a message that is sent by the "HandleAwsDataPlan" and contains TAF and AWS wind speed data. This plan also sends out the message "AlertDiscrepancy" to the "Alerter" agent if a discrepancy is detected.

The change actions presented ahead are also the actual changes that were made by all participants in the user study evaluation performed in [Jayatilleke, 2007]. It is noted that these actions are presumably performed by the designer using PDT, which may and may not map directly to the metamodel on a one-to-one basis:

- Linking and unlinking between two entities in PDT corresponds to connection and disconnection respectively. For instance, in PDT linking "HandleAwsDataPlan" plan

influence on the change propagation process.

[11] In fact the omission of wind speed handling in the design was not accidental, but we are viewing it as accidental.

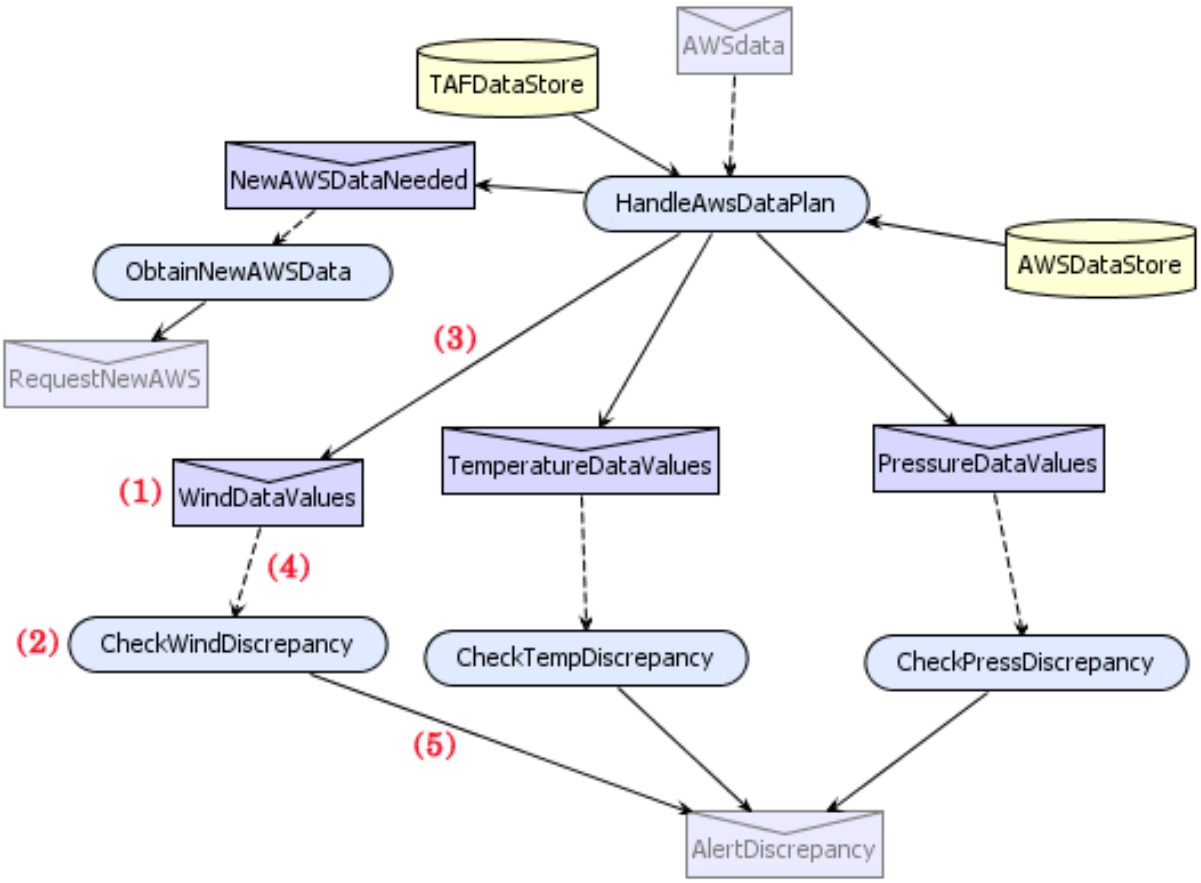


Figure 9.9: The agent overview diagram for "Discrepancy" agent after change 1

to "WindDataValues" message (send) means that make the plan send the message. With respect to the metamodel, this action is actually connecting a Plan entity named 'HandleAwsDataPlan" plan and a Message entity named "WindDataValues" using the *planSender–outgoingMessagesEntityReference* association (refer to the Prometheus metamodel in figure 4.16 on page 92).

- Creating an entity in another entity map corresponds to *two* changes to the metamodel. For instance, creating "WindDataValues" message in "Discrepancy" agent implies creating "WindDataValues" message and linking it with the "Discrepancy" agent so that it is owned by the agent (following the *internalMessage–agentOwner* association, refer to figure 4.12 on page 87).

- Making a percept or message to be a trigger of a plan corresponds to connecting the

percept or message to the plan with respect to the *triggersEntityReferencePercept–triggeredPlan* or *triggersEntityReferenceMessage–triggeredPlan* association in the meta-model (refer to figure 4.16 on page 92).

Below are the specific change actions[12], depicted in figure 9.9[13].

1. Create "WindDataValues" message in "Discrepancy" agent.
2. Create "CheckWindDiscrepancy" plan in "Discrepancy" agent.
3. Link "HandleAwsDataPlan" plan to "WindDataValues" message (send).
4. Make "WindDataValues" message to be a trigger of "CheckWindDiscrepancy" plan.
5. Link "CheckWindDiscrepancy" plan to "AlertDiscrepancy" message (send).

Result

The following sequence of actions captures the interaction between the "simulated" designer and the tool in terms of what actions were performed by the designer and what actions were proposed and executed by the tool.

1. Create "WindDataValues" message in "Discrepancy" agent (done by the designer).
   After this, the tool identifies several violated constraints such as the new messages is not sent and received by any plans. The tool then returns 9 options, one of which matches the designer's plan, resulting in the next three steps being done by the tool.
2. Create "CheckWindDiscrepancy" plan in "Discrepancy" agent *(done by the tool)*[14].
3. Link "HandleAwsDataPlan" plan to "WindDataValues" message (send) *(done by the tool).*
4. Make "WindDataValues" message to be a trigger of "CheckWindDiscrepancy" plan *(done by the tool).*
   After step 4 is done, the design is consistent: linking "CheckWindDiscrepancy" plan to "AlertDiscrepancy" message is not needed since the new message already has its sender and receiver, and the new plan already has its trigger.

[12]Some variations in order are possible, but they do not affect the outcome.
[13]Each number in the figure corresponds to a relevant step in the sequence of change actions.
[14]The tool creates a new plan and asks the designer to provide a name for the new plan.

5. Link "CheckWindDiscrepancy" plan to "AlertDiscrepancy" message (send) (done by the designer).

This change is relatively straightforward and it does not involve any side-effect actions. The tool is able to pick up three steps out of five steps contained in the change plan.

9.5.2 Change 2: Implementing a variable threshold alerting

Currently, the alerting thresholds are fixed and hard-coded for each data type (e.g. 2 for pressure, 5 for temperature, and 10 for wind). For example, if a discrepancy between TAF and AWS values for pressure is above 2, in any of the regions (e.g. Melbourne, Sydney and Darwin), a warning is generated. However, the forecast personnel wants to be able to adjust the alerting threshold levels. More specifically, different regions will show alerts based on different discrepancies, for example in the Melbourne region the pressure threshold might be 3 whereas for Sydney might be 6. Such values are provided and can be changed by the user (e.g. airport weather manager) in each region. Hence, a new requirement that the forecast personnel should be able to set a threshold for alerting is requested. This requirement change involves modification of the existing alerting functionality to respond to a business change. Therefore, we place it in the category of perfective and modification of our change taxonomy.

This change request implies two major design changes: (1) alerting threshold levels should be realised as the user's input, and (2) these data should be stored and used when discrepancy detection takes place. User inputs are usually represented as percepts. The current design has only three percepts and each of them has its own purpose: "TAF" contains data input from the forecaster, "AWS" is input from the automatic weather sensor and "Change Subscription" is requested from the airport official to subscribe/unsubscribe to certain information. Therefore, a new percept is needed to represent the user's request for changing the thresholds. This percept also contains a new threshold for either temperature or pressure.

Since the "GUI" agent is responsible for interacting with the system's users, it should be extended to handle this new percept and the agent thus needs a new plan which responds to this percept (it can use existing plans but this would violate the cohesion principle). This plan processes the alerting threshold input contained in the percept and stores it in a data store. Because existing data stores are not for keeping the threshold information, a new data store is needed. This data store is written by the "GUI" agent and its new plan. Finally, as the "Discrepancy" agent is responsible for detecting discrepancies, it should be changed to meet

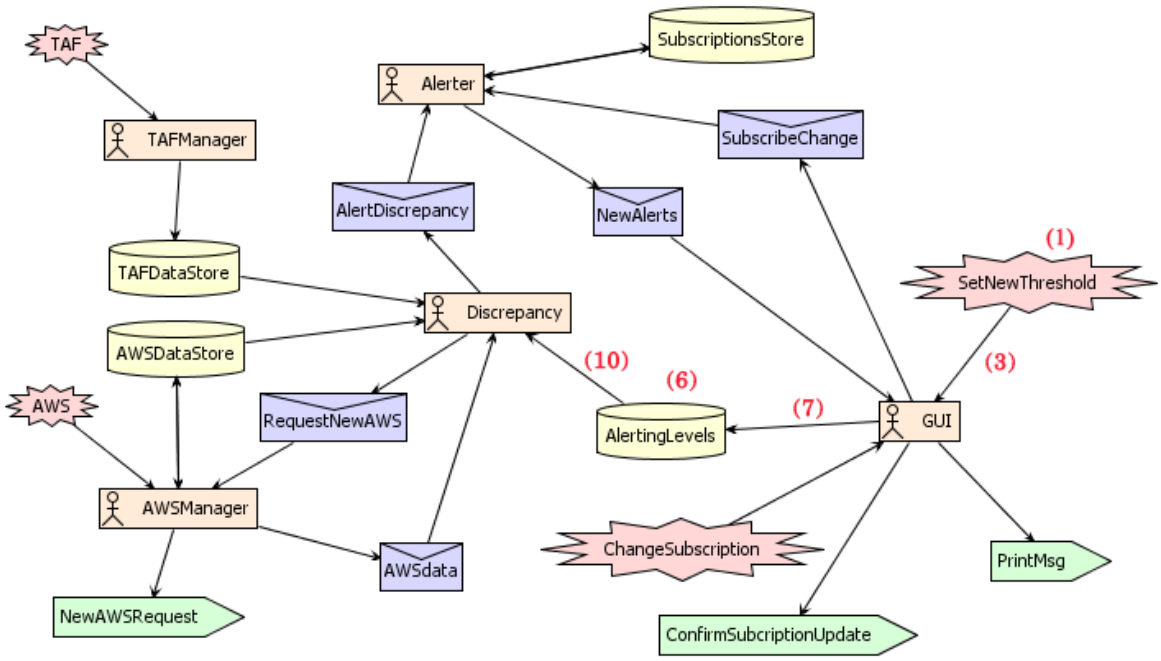


Figure 9.10: The system overview diagram after change 2

the second design change requirement. More specifically, the "Discrepancy" agent's plans for detecting discrepancies, i.e. "CheckTempDiscrepancy" and "CheckPressDiscrepancy", will need to use the new threshold data store to determine if a discrepancy warning should be raised.

The above changes are made to the detailed design artefacts, and result in several side-effects (see section 9.3 for a definition of side-effects) in architectural and system specification artefacts. First, the new percept needs to be handled by a role played by the "GUI" agent. Second, the new data should be written by a role played by the "GUI" agent and read by a role played by "Discrepancy" agent.

The designer's complete plan D thus consists of the following sequence[15]. Figures 9.10 and 9.11 shows the system overview diagram and the agent overview diagram for "GUI" agent after these changes are made. Note that changes are also made to other diagrams (e.g. side-effect actions are shown in the system roles diagram and the data coupling diagram, and action steps 12 and 13 are done in the agent overview diagram for "Discrepancy" agent (not shown)).

[15]Some variations in order are possible, but they do not affect the outcome of the evaluation.

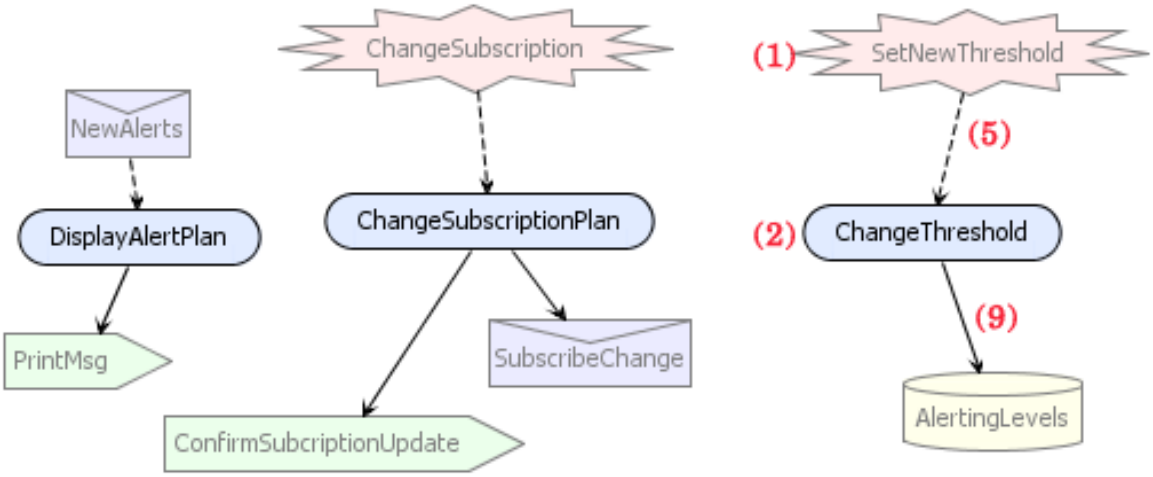


Figure 9.11: The agent overview diagram for "GUI" agent after change 2

1. Create "SetNewThreshold" percept.
2. Create "ChangeThreshold" plan in "GUI" agent.
3. Link "SetNewThreshold" percept to "GUI" agent.
* 4. Link "SetNewThreshold" percept to "User Interaction" role[16].
5. Make "SetNewThreshold" percept to be a trigger of "ChangeThreshold" plan.
6. Create a new "AlertingLevels" data.
7. Link "GUI" agent to "AlertingLevels" data (write.)
* 8. Link "User Interaction" role to "AlertingLevels" data (write).
9. Link "ChangeThreshold" plan to "AlertingLevels" data (write).
10. Link "AlertingLevels" data to "Discrepancy" agent (read).
*11. Link "AlertingLevels" data to "Check Discrepancies" role (read).
12. Link "AlertingLevels" data to "CheckTempDiscrepancy" plan (read)[17].
13. Link "AlertingLevels" data to "CheckPressDiscrepancy" plan (read)[17].

[16]Side effect actions are marked with an asterisk (*) next to their numbers.
[17]This action takes place in the agent overview diagram for agent "Discrepancy" (not shown).

It is noted that changes should also be made to a plan's internals (e.g. "CheckTempDiscrepancy" plan and "CheckPressDiscrepancy" plan). However, such changes are not made at the design level, but at the code level[18].

Result

The following sequence of actions captures the interaction between the "simulated" designer and the tool in terms of what actions were performed by the designer and what actions were proposed and executed by the tool.

1. Create "SetNewThreshold" percept (done by the designer).
   After this, the tool identifies three violated constraints since the new percept needs to be handled by a role, an agent and a plan. The tool then returns 18 repair options, one of which matches the designer's plan, resulting in the next four steps being done by the tool.

2. Create "ChangeThreshold" plan in "GUI" agent *(done by the tool)*[19].

3. Link "SetNewThreshold" percept to "GUI" agent *(done by the tool).*

* 4. Link "SetNewThreshold" percept to "User Interaction" role[20] *(done by the tool).*

5. Make "SetNewThreshold" percept to be a trigger of "ChangeThreshold" plan *(done by the tool).*

6. Create a new "AlertingLevels" data (done by the designer).
   After this, the tool identifies violated constraints relating to the requirements of a new data being needed to be accessed (either written or read) by an agent and plan. The tool then returns 26 options, three of which match the designer's plan. The first option suggests steps 7-9, the second option recommends steps 10-12, and the third option suggests steps 10, 11, and 13. The designer chooses one of these options, for example the first option, resulting in the next 3 steps being done by the tool[21].

[18] In PDT, such changes can be made in the plan descriptor's body section in terms of specifying pseudo code.

[19] The tool creates a new plan and asks the designer to provide a name for the new plan.

[20] Side effect actions are marked with an asterisk (*) next to their numbers.

[21] If the designer could select and execute more than one option at a time then in this particular change case, the tool is even more effective: the designer could select all three matching options, and consequently have all the remaining steps (7-13) be done by the tool.

7. Link "GUI" agent to "AlertingLevels" data (write) *(done by the tool).*

* 8. Link "User Interaction" role to "AlertingLevels" data (write) *(done by the tool).*

9. Link "ChangeThreshold" plan to "AlertingLevels" data (write) *(done by the tool).* After this step, the design is consistent because the new data is now accessed by an agent, a plan and a role. Therefore, the next step is done by the designer.

10. Link "AlertingLevels" data to "Discrepancy" agent (read) (done by the designer). After this, the tool returns 3 options, two of which match the designer plans. The first option suggests steps 11 and 12, whilst the second option suggests steps 11 and 13. The designer chooses one of these options, and manually performs the remaining step.

*11. Link "AlertingLevels" data to "Check Discrepancies" role (read) *(done by the tool).*

12. Link "AlertingLevels" data to "CheckTempDiscrepancy" plan (read) *(done by the tool).*

13. Link "AlertingLevels" data to "CheckPressDiscrepancy" plan (read) (done by the designer).

The tool is able to identify and perform 9 actions out of 13 actions (including 3 side-effect actions) in the change plan.

9.5.3 Change 3: Adding volcanic ash

One of the major updates made to the original application trialled at the Bureau of Meteorology was the inclusion of volcanic ash alerts. A motivation behind this requirement change was the availability of information on volcanic activities reported on special mailing lists. In addition, apart from reporting on discrepancies between AWS and TAF data, it is useful to add the ability to generate other forms of weather alerts related to airports. One such alert is on volcanic ash effect due to volcanic activities in the vicinity. By subscribing to these mailing lists, one could obtain data about possible volcanic ash effects on a given area, where volcanic activity in the vicinity has occurred. Since this requirement change involves the addition of a new functionality (i.e. alerts on volcanic ash) due to a change in the operational environment (i.e. the availability of volcanic ash data), we classify this change into the category of adaptive and addition in our taxonomy.

The change required the addition of a new percept (i.e. "VolcanicAshEmail") to represent the volcanic ash related data. Although we could in theory use existing agents (e.g. "TAFManager") for reading volcanic ash data and generating alerts when the ash levels are above a certain limit, this violates modularity and cohesion principles. Therefore, we prefer to create a new agent (i.e. "VolcanicManager") to handle the new percept. Similarly, this agent should contain a new plan which handles this percept. For debugging purposes and/or future use, a new data store ("VolcanicAshData") is created to keep the processed volcanic ash data. This data store is written by the new agent and its only plan (i.e. "ProcessVolcanicAshDataPlan").

Similar to the normal alerts, volcanic alerts should be sent to subscribed GUIs by the "Alerter" agent. Therefore, the "VolcanicManager" agent should notify the "Alerter" agent with a message ("AlertVolcanicAshLevel") which will trigger the process of sending out volcanic alerts to appropriate "GUI" agents. Currently, the "Alerter" has two plans and both of them are already triggered by other messages. Hence, a new plan is needed to handle the incoming message related to volcanic ash levels. This plan also uses data in the "SubscriptionsStore" to work out the subscribed GUIs which should receive volcanic alerts. Changes are also needed in the "NewAlerts" and "SubscribeChange" messages so that they include volcanic ash related data and subscription respectively. However such changes should be made at the implementation level, and thus we do not include them here[22].

Below are the specific change actions[23]. Figures 9.12, 9.13 and 9.14 show the main action steps.

1. Create "VolcanicManager" agent.

* 2. Create "Manage Volcanic Ash" role.

* 3. Link "VolcanicManager" agent to "Manage Volcanic Ash" role.

4. Create "VolcanicAshEmail" percept.

5. Create "ProcessVolcanicAsh" plan in "VolcanicManager" agent.

6. Link "VolcanicAshEmail" percept to "VolcanicManager" agent.

* 7. Link "VolcanicAshEmail" percept to "Manage Volcanic Ash" role.

[22]In PDT, such changes can be made in the message descriptor in terms of specifying text description.
[23]Some variations in order are possible, but they do not affect the outcome.

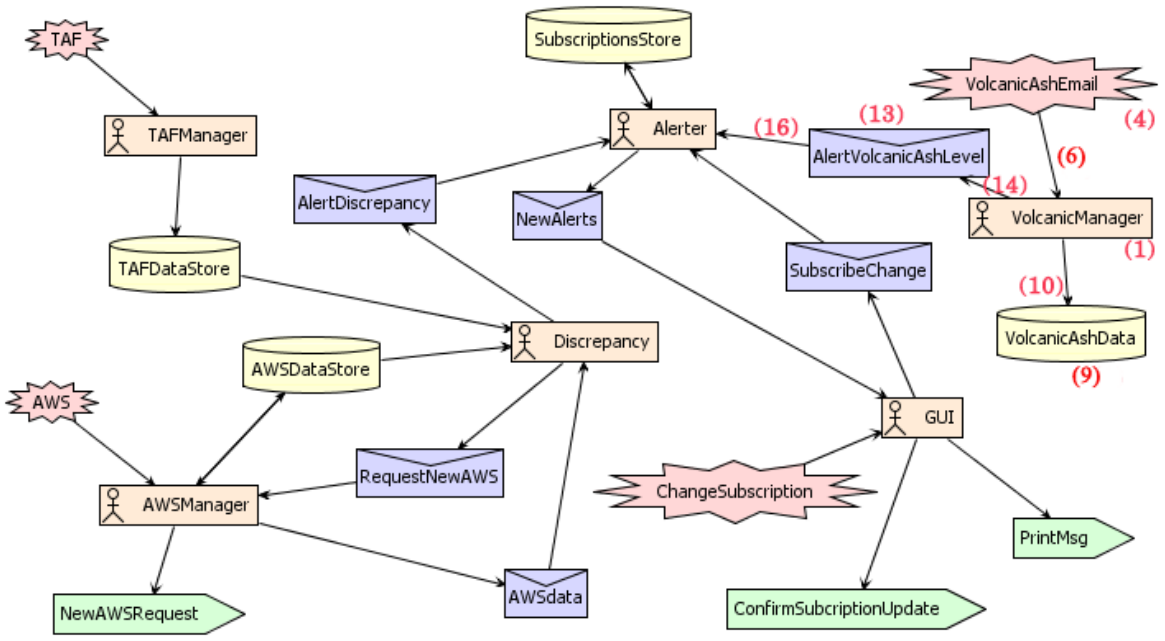


Figure 9.12: The system overview diagram after change 3

8. Make "VolcanicAshEmail" percept to be a trigger of "ProcessVolcanicAsh" plan.
9. Create "VolcanicAshData" data.
10. Link "VolcanicManager" agent to "VolcanicAshData" (write).
*11. Link "Manage Volcanic Ash" role to "VolcanicAshData" (write).
12. Link "ProcessVolcanicAsh" plan to "VolcanicAshData" (write).
13. Create "AlertVolcanicAshLevel" message.
14. Link "VolcanicManager" agent to "AlertVolcanicAshLevel" message (send).
15. Link "ProcessVolcanicAsh" plan to "AlertVolcanicAshLevel" message (send).
16. Link "AlertVolcanicAshLevel" message to "Alerter" agent (receive).
17. Create "SendVolcanicAlertToSubscribedGUIs" plan in "Alerter" agent
18. Make "AlertVolcanicAshLevel" message to be a trigger of "SendVolcanicAlertToSubscribedGUIs" plan.

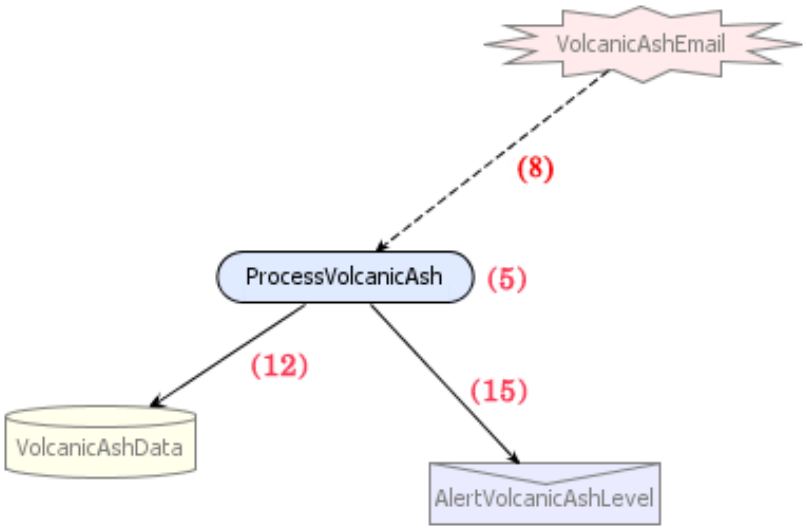


Figure 9.13: The agent overview diagram for "VolcanicManager" agent after change 3

19. Link "SendVolcanicAlertToSubscribedGUIs" plan to "NewAlerts" message (send).
20. Link "SubscriptionsStore" data to "SendVolcanicAlertToSubscribedGUIs" plan (read).

Result

The following sequence of actions captures the interaction between the "simulated" designer and the tool in terms of what actions were performed by the designer and what actions were proposed and executed by the tool.

1. Create "VolcanicManager" agent (done by the designer).
   After this, the tool identifies a violated constraint: the new agent needs to play a role. The tool returns one option that matches the designer's plan, resulting in the next two steps is done by the tool.

* 2. Create "Manage Volcanic Ash" role *(done by the tool)*[24].

* 3. Link "VolcanicManager" agent to "Manage Volcanic Ash" role *(done by the tool)*.

4. Create "VolcanicAshEmail" percept (done by the designer).
   After this, the tool identifies three violated constraints since the new percept needs to

[24]The tool creates a new role and asks the designer to provide a name for the new role.

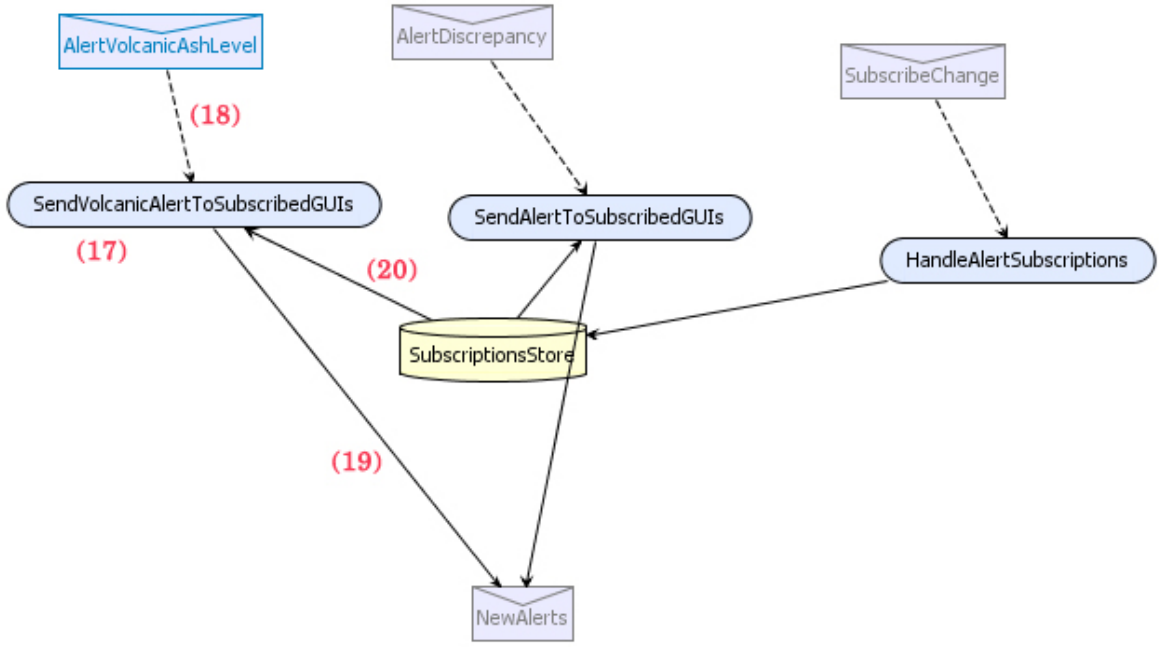


Figure 9.14: The agent overview diagram for "Alerter" agent after change 3

be handled by a role, an agent and a plan. The tool then returns 19 change options, one of which matches the designer's plan, resulting in the next four steps being done by the tool.

5. Create "ProcessVolcanicAsh" plan in "VolcanicManager" agent *(done by the tool)*[25].

6. Link "VolcanicAshEmail" percept to "VolcanicManager" agent *(done by the tool).*

* 7. Link "VolcanicAshEmail" percept to "Manage Volcanic Ash" role *(done by the tool).*

8. Make "VolcanicAshEmail" percept to be a trigger of "ProcessVolcanicAsh" plan *(done by the tool).*

9. Create "VolcanicAshData" data (done by the designer).
   After this, the tool returns 26 options, one of which matches the designer's plan, resulting in the next three steps being done by the tool.

10. Link "VolcanicManager" agent to "VolcanicAshData" (write) *(done by the tool).*

[25]The tool creates a new plan and asks the designer to provide a name for the new plan.

*11. Link "Manage Volcanic Ash" role to "VolcanicAshData" (write) *(done by the tool).*

12. Link "ProcessVolcanicAsh" plan to "VolcanicAshData" (write) *(done by the tool).*

13. Create "AlertVolcanicAshLevel" message (done by the designer).
    After this, the tool returns 25 options but none of them matched the change plan. This can be explained by the fact that the tool presented only cheapest options, which in this case is making the new message internal to an existing agent (consequently saving the cost of making a link between an agent and the new message). The intended change plan, in contrast, makes the new message external to all the agents.

14. Link "VolcanicManager" agent to "AlertVolcanicAshLevel" message (send) (done by the designer).
    After this, the tool returns 18 options, one of them matches the change plan, resulting in the next four steps being done by the tool.

15. Link "ProcessVolcanicAsh" plan to "AlertVolcanicAshLevel" message (send) *(done by the tool).*

16. Link "AlertVolcanicAshLevel" message to "Alerter" agent (receive) *(done by the tool).*

17. Create "SendVolcanicAlertToSubscribedGUIs" plan in "Alerter" agent *(done by the tool).*

18. Make "AlertVolcanicAshLevel" message to be a trigger of "SendVolcanicAlertToSubscribedGUIs" plan *(done by the tool).*
    After this, the model becomes consistent, i.e. the next two steps are not needed for consistency. Therefore, these steps have to be done by the designer.

19. Link "SendVolcanicAlertToSubscribedGUIs" plan to "NewAlerts" message (send) (done by the designer).

20. Link "SubscriptionsStore" data to "SendVolcanicAlertToSubscribedGUIs"plan (read) (done by the designer).

The tool performs 13 out of 20 actions in the change plan. All the four side-effect actions are also identified and performed by the tool.

9.5.4 Change 4: Logging sent alerts

As a means of tracking alerts generated, and also for verification purposes, a new requirement is that all alerting messages sent to the GUI component of the system should be logged. This change can be classified as perfective and addition of new functionality.

Since we need to store the data being logged, a new data store (called "AlertLogData") is introduced. At the moment, the "Discrepancy" agent detects significant differences between TAF data and AWS data and notifies the "Alerter" agent when this situation happens. The "Alerter" agent is responsible for handling alerts and sending them to the registered "GUI" agents as well as managing the alert subscriptions. Hence, there are two likely design alternatives to meet this change request, corresponding to either the "Discrepancy" agent or the "Alerter" agent being responsible for handling alert logging. Within the "Discrepancy" agent, the two plans "CheckTempDiscrepancy" and "CheckPressDiscrepancy" notify the "Alerter" agent when discrepancies happen. As a result, to meet the change request they should be modified. Similarly, the "SendAlertToSubscribedGUIs" plan should be modified if the "Alerter" agent is chosen to handle alert logging.

Below are the list of change actions that correspond to the first design alternative[26]. Figure 9.15 shows how changes are made to the agent overview diagram for the "Alerter" agent. It is noted that the action step 3, i.e. link "Manage Subscriptions" role to "AlertLog" data, is considered a side-effect action and should be performed in the role diagram which we do not show here.

`Alternative 1: modifying ''Alerter'' agent`

1. Create "AlertLog" data.
2. Link "Alerter" agent to "AlertLog" data (write).

* 3. Link "Manage Subscriptions" role to "AlertLog" data (write).

4. Link "SendAlertToSubscribedGUIs" plan to "AlertLog" data (write).

Below are the list of change actions that correspond to the second design alternative[27]. Figure 9.16 shows how changes are made to the agent overview diagram for the "Discrepancy" agent. It is noted that the action step 3, i.e. link "Check Discrepancy" role to "AlertLog"

[26] Some variations in order are possible, but they do not affect the outcome.
[27] Some variations in order are possible, but they do not affect the outcome.

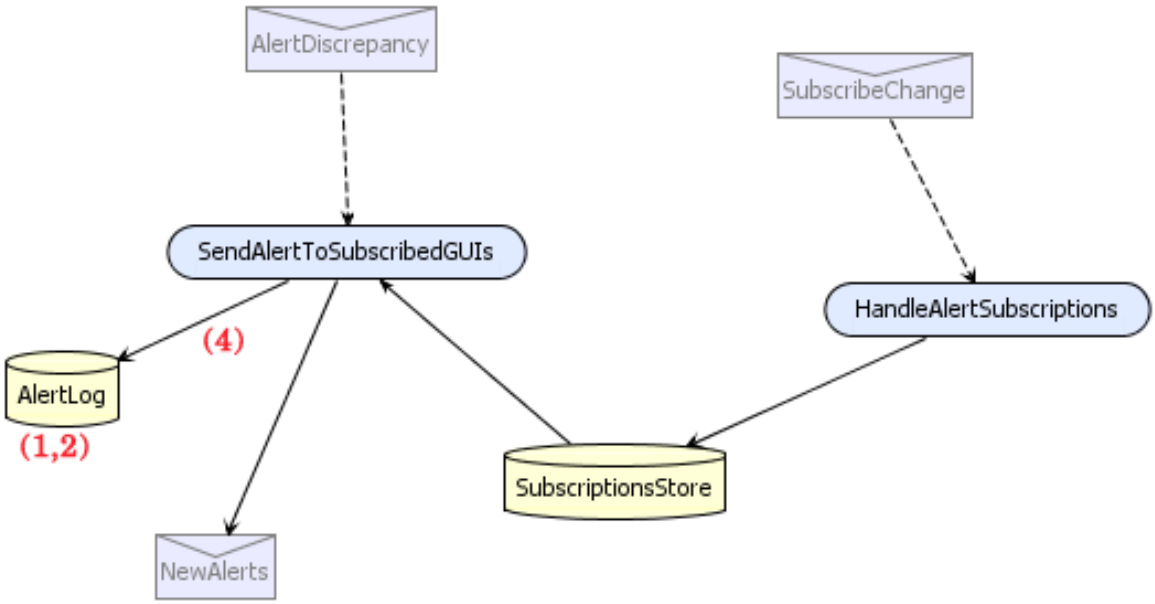


Figure 9.15: The agent overview diagram for "Alerter" agent after change 4

data, is considered a side-effect action and should be performed in the role diagram which we do not show here.

`Alternative 2: modifying ‘‘Discrepancy’’ agent`

1. Create "AlertLog" data.
2. Link "Discrepancy" agent to "AlertLog" data (write).

* 3. Link "Check Discrepancy" role to "AlertLog" data (write).

4. Link "CheckTempDiscrepancy" plan to "AlertLog" data (write) .
5. Link "CheckPressDiscrepancy" plan to "AlertLog" data (write).

Result

There are two alternative change plans to meet this change requirement. However, they share the same first step, i.e. creating "AlertLog" data. The tool responds with 24 options, which include matches for each of the two alternatives. However, if the designer decides to go with alternative 2, he/she needs to manually perform either step 4 or 5. It is also noted that the tool does pick up the side-effect action in each alternative.

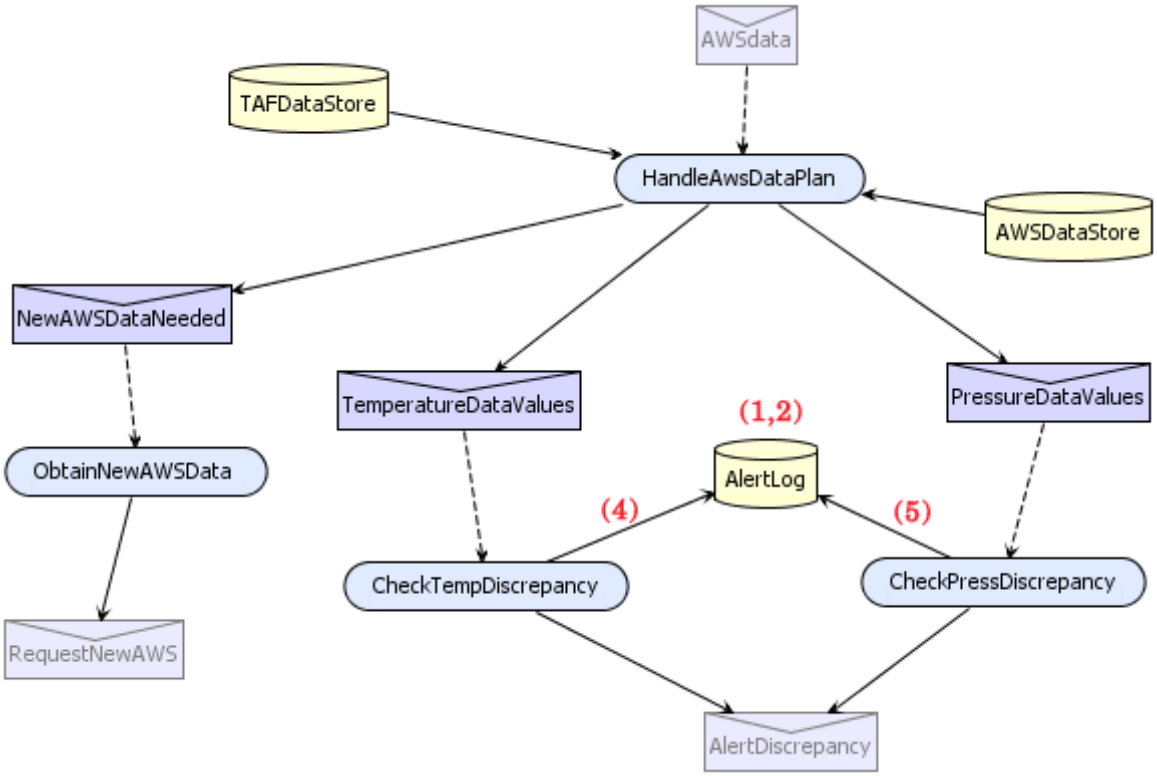


Figure 9.16: The agent overview diagram for "Discrepancy" agent after change 4

`Alternative 1: modifying ''Alerter'' agent`

1. Create "AlertLog" data (done by the designer).
   After this, the tool identifies violated constraints relating to the requirements of a new data being needed to be accessed by an agent, plan, and role. The tool then returns 24 options, resulting in the next 3 steps being done by the tool.
2. Link "Alerter" agent to "AlertLog" data (write) *(done by the tool).*

* 3. Link "Manage Subscriptions" role to "AlertLog" data (write) *(done by the tool).*

4. Link "SendAlertToSubscribedGUIs" plan to "AlertLog" data (write) *(done by the tool).*

`Alternative 2: modifying ''Discrepancy'' agent`

1. Create "AlertLog" data (done by the designer).
   Similarly to the first alternative, the tool returns with 24 options, resulting in the next 2 steps and either step 4 or 5 being done by the tool.

2. Link "Discrepancy" agent to "AlertLog" data (write) *(done by the tool).*

* 3. Link "Check Discrepancy" role to "AlertLog" data (write) *(done by the tool).*

4. Link "CheckTempDiscrepancy" plan to "AlertLog" data (write) *(done by the tool).* After step 4 is done the design is consistent: linking "CheckPressDiscrepancy" plan to "AlertLog" data is not needed for restoring consistency because the new data has been accessed by a plan.

5. Link "CheckPressDiscrepancy" plan to "AlertLog" data (write) (done by the designer).

9.5.5 Change 5: Having multiple "TAF Manager" agents

The initial system has only one "TAFManager" agent instance to deal with forecast data from different airports. As the number of airports participating in the system increases, it is required to have multiple TAF Manager agents (instead of only one) to better deal with the load. This requirement results from a change in the operational environment (i.e. the number of airports growing), which may require a modification of existing functionality. Therefore, we classify this change into the category of adaptive and modification.

With the current design, all "TAF" percepts are dealt with by a single "TAFManager" agent, which processes "TAF" data and keeps them in a central "TAFDataStore". The "Discrepancy" agent accesses this data store to retrieve "TAF" data. This design is suitable if there is only one "TAFManager" agent instance. However, we now need to have multiple "TAFManager" agent instances and we do not want them all trying to write to one data store. Instead, we want each of the "TAFManager" agents to send TAF data to the "Discrepancy" agent, which will manage that "TAFDataStore".

Although the above set of actions successfully meets the new requirement change, there are still actions that need to be done to make the detailed design consistent with the specification design. More specifically, changes have to be made to the scenario and roles as noted below.

Below is the list of change actions[28]. Figures 9.17, 9.18, 9.19, and 9.20 depict some of the main action steps.

1. Unlink "ProcessTAF" plan and "TAF Data Store" data

[28]Some variations in order are possible, but they do not affect the outcome.

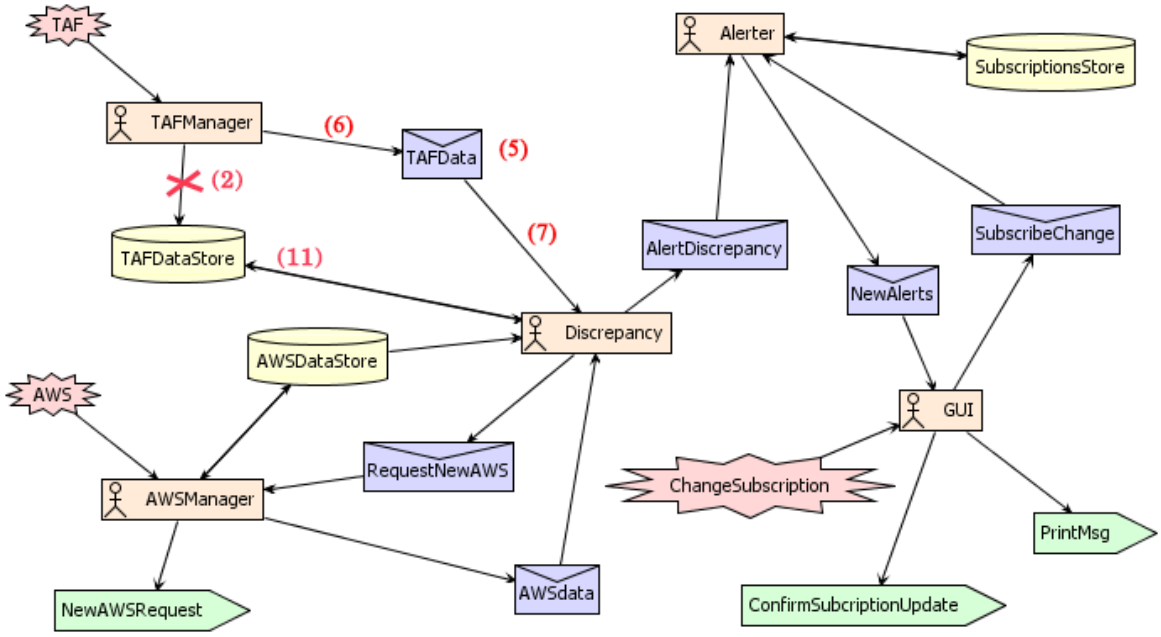


Figure 9.17: The system overview diagram after change 5

2. Unlink "TAF Manager" agent and "TAF Data Store" data.

* 3. Unlink "Manage TAF Data" role and "TAF Data Store" data.

* 4. Unlink "TAF Data Store" data and the last step in "Process TAF" scenario.

5. Create "TAFData" message.

6. Link "TAF Manager" agent with "TAFData" message (send).

7. Link "TAFData" message with "Discrepancy" agent (receive).

8. Link "ProcessTAF" plan with "TAFData" message (send).

9. Create "HandeTafDataPlan" plan in "Discrepancy" agent.

10. Link "TAF Data" message with "HandeTafDataPlan" plan (trigger).

11. Link "Discrepancy" agent with "TAF DataStore" (write)[29].

*12. Link "Check Discrepancies" role with "TAF Data Store" (write).

[29]Note that the "Discrepancy" agent already reads the "TAF DataStore"

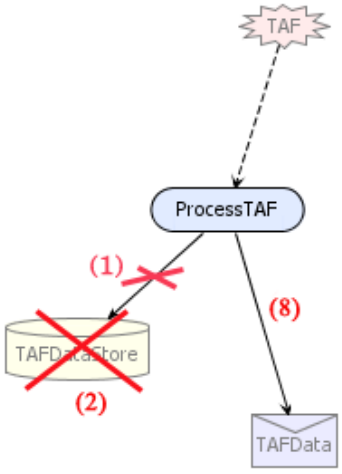


Figure 9.18: The agent overview diagram for "TAFManager" agent after change 5

13. Link "HandeTafDataPlan" plan with "TAF Data Store" (write).

Result

This change scenario is triggered by a disconnection, which indicates further disconnections are required. Therefore, the designer decides to encourage disconnection within the change propagation process. He/she does so by setting the cost of disconnection be equal the cost of connection (i.e. cost of 1).

The following sequence of actions captures the interaction between the "simulated" designer and the tool in terms of what actions were performed by the designer and what actions were proposed and executed by the tool.

1. Unlink "ProcessTAF" plan and "TAF Data Store" data (done by the designer).
   After this, the tool identifies an inconsistency between plan "ProcessTAF" and its owning agent "TAFManager" in terms of accessing "TAF Data Store". The tool then returns one option: linking "ProcessTAF" plan and "TAF Data Store" data again, which does not match the designer's plan.

2. Unlink "TAF Manager" agent and "TAF Data Store" data (done by the designer).
   After this, the tool identifies an inconsistency between the "TAF Manager" agent and its playing role "Manage TAF Data". The tool then returns two options, one of which matches the designer's plan resulting in the next two steps being done by the tool.

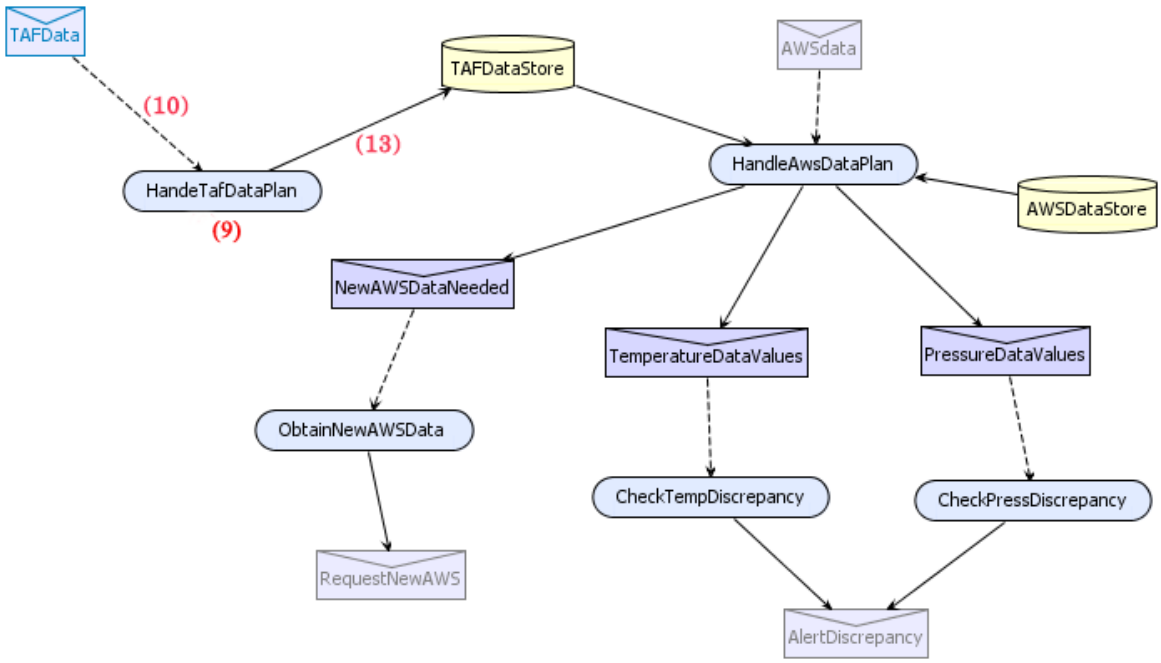


Figure 9.19: The agent overview diagram for "Discrepancy" agent after change 5

* 3. Unlink "Manage TAF Data" role and "TAF Data Store" data *(done by the tool).*

* 4. Unlink "TAF Data Store" data and the last step in "Process TAF" scenario *(done by the tool).*

5. Create "TAFData" message (done by the designer).
   After this, the tool returns 27 options, none of which match the change plan because the cheapest options make the message an internal message of an existing agent.

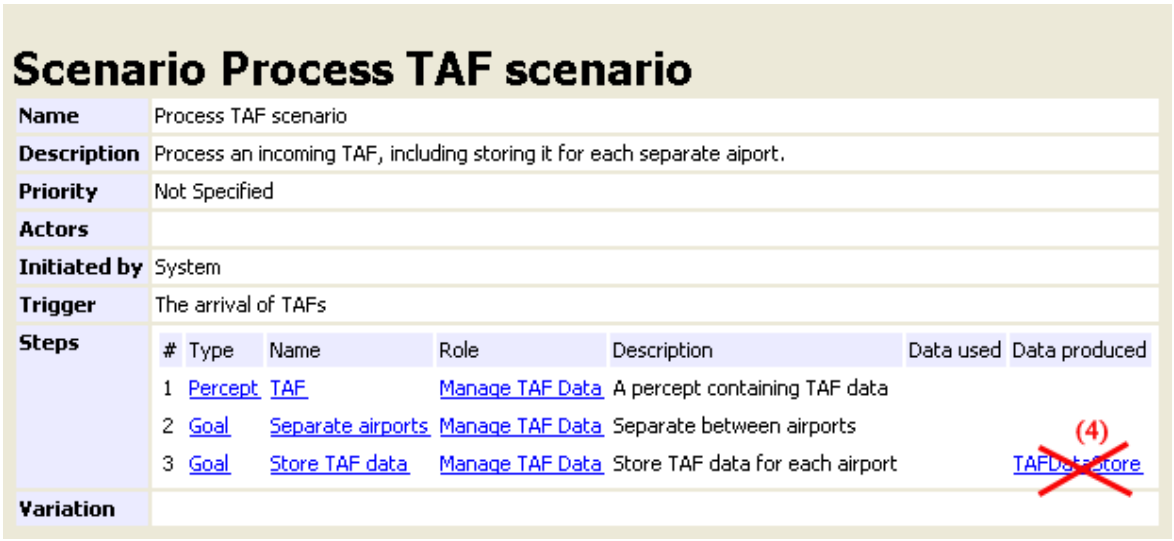
6. Link "TAF Manager" agent with "TAFData" message (send) (done by the designer).
   After this, the tool returns 18 options, one of them matches the change plan, resulting in the next four steps being done by the tool.

7. Link "TAFData" message with "Discrepancy" agent (receive) *(done by the tool).*

8. Link "ProcessTAF" plan with "TAFData" message (send) *(done by the tool).*

9. Create "HandeTafDataPlan" plan in "Discrepancy" agent *(done by the tool).*

10. Link "TAF Data" message with "HandeTafDataPlan" plan (trigger) *(done by the tool).*



Scenario Process TAF scenario

Name	Process TAF scenario
Description	Process an incoming TAF, including storing it for each separate aiport.
Priority	Not Specified
Actors	
Initiated by	System
Trigger	The arrival of TAFs

Steps

#	Type	Name	Role	Description	Data used	Data produced
1	Percept	TAF	Manage TAF Data	A percept containing TAF data		
2	Goal	Separate airports	Manage TAF Data	Separate between airports		
3	Goal	Store TAF data	Manage TAF Data	Store TAF data for each airport		TAFDataStore

(4)

Variation

Figure 9.20: The "Process TAF" scenario after change 5

11. Link "Discrepancy" agent with "TAF Data Store" (write)[30] (done by the designer). The tool then returns 4 options, one of which contains all the remaining steps.

*12. Link "Check Discrepancies" role with "TAF Data Store" (write) *(done by the tool).*

13. Link "HandeTafDataPlan" plan with "TAF Data Store" (write) *(done by the tool).*

Of 13 actions in the change plan, 8 actions have been done by the tool. In addition, the tool is able to identify and perform all three side-effect actions.

9.5.6 Change 6: Subscription

The initial system only alerted on discrepancies between weather readings (AWS) and forecasts (TAF). Forecast information is, however, important to pilots especially from the regions that they are about to fly in. As a result, this change requires the necessary modifications to be made to allow the pilots (i.e. the "GUI" agent) to have direct notification of TAF data changes in the regions that they are interested in. We consider this change as perfective and addition maintenance in our taxonomy.

As the "TAF Manager" agent is responsible for processing TAF data, we make it send a "TAFData" message to notify the "GUI" agent. Furthermore, the internals of those two

[30]Note that the "Discrepancy" agent already reads the "TAF DataStore".

agents need modifications. The "ProcessTAF" plan in the "TAFManager" agent now also sends a "TAFData" message. A new plan is created within the "GUI" agent to receive this message.

The above changes are the main actions. One important set of side-effect actions that the designer tends to miss is to make the "GUI" agent subscribe with the "TAF Manager" agent. This is to make sure that the "TAFData" message is sent to the right GUI agent instance.

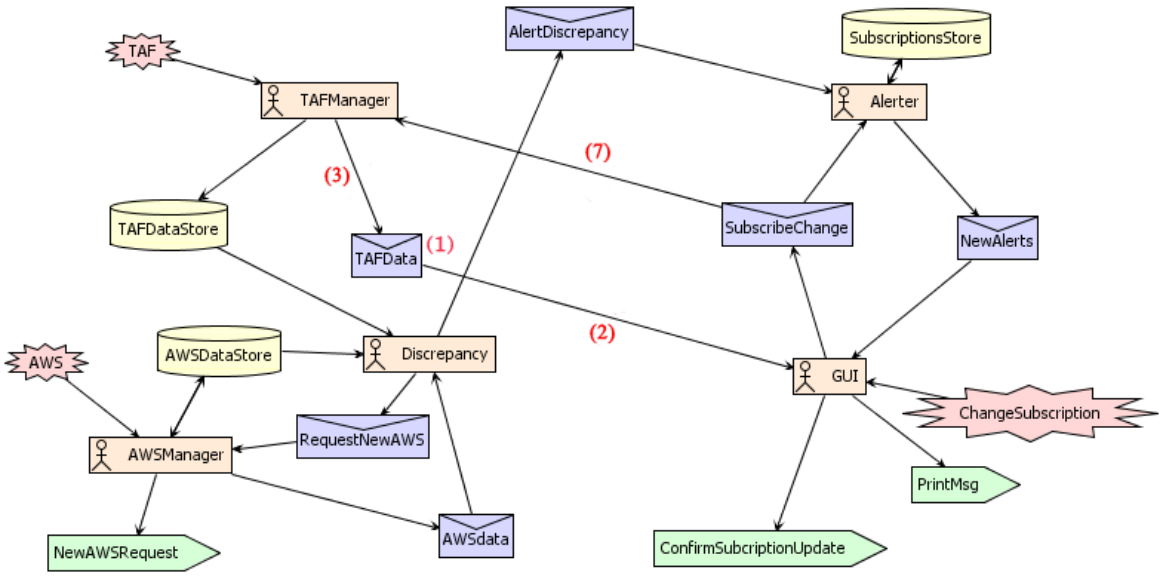


Figure 9.21: The system overview diagram after change 6

Below is the list of change actions. Figures 9.21 and 9.22 depicts some of the main action steps, including the side-effect ones.

1. Create "TAFData" message.
2. Link "TAFData" message with the "GUI" agent (receive).
3. Link "TAF Manager" agent with "TAFData" message (send).
4. Link "ProcessTAF" plan with "TAFData" message (send).
5. Create a "Use TAFData" plan in "GUI" agent[31].
6. Make "TAFData" message to be a trigger of "Use TAF Data" plan[31].

[31]This action takes place in the agent overview diagram for agent "GUI" (not shown).

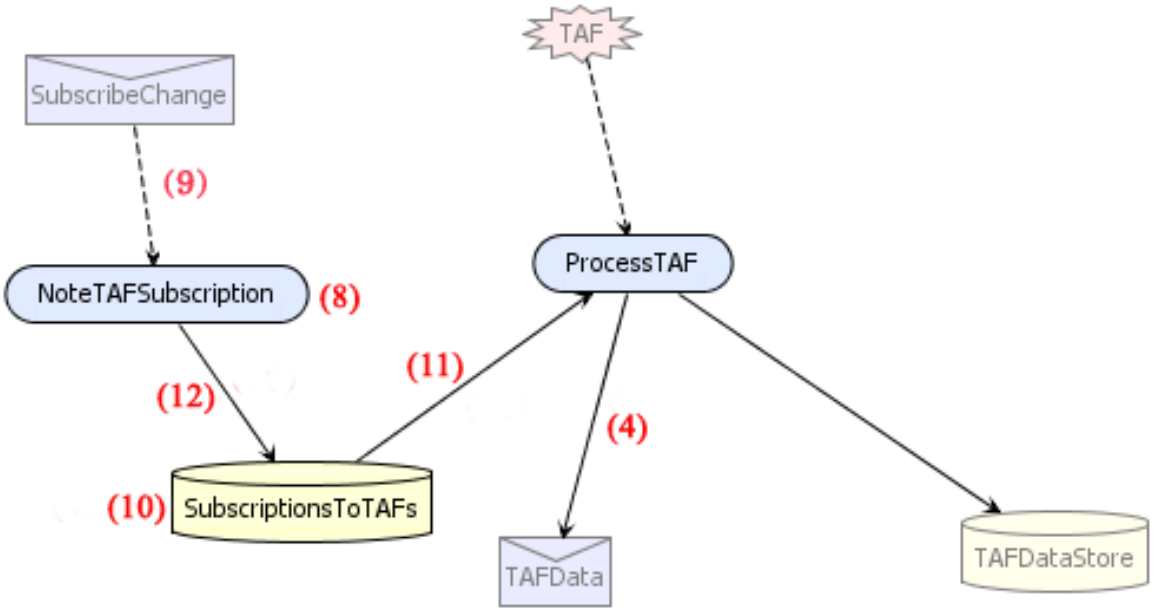


Figure 9.22: The agent overview diagram for "TAF Manager" agent after change 6

* 7. Link "SubscribeChange" message with "TAFManager" agent (receive).

* 8. Create "NoteTAFSubscription" plan in "TAFManager" agent.

* 9. Make "SubscribeChange" message to be a trigger of "NoteTAFSubscription" plan (receive).

*10. Create "SubscriptionsToTAFs" data in "TAFManager" agent.

*11. Link "SubscriptionsToTAFs" data with "ProcessTAF" (read).

*12. Link "NoteTAFSubscription" plan with "SubscriptionsToTAFs" data (write).

*13. Link "SubscriptionsToTAFs" data "Manage TAF Data" role to (read)[32].

*14. Link "Manage TAF Data" role to "SubscriptionsToTAFs" data (write)[32].

Result

1. Create "TAFData" message (done by the designer).
   The tool returns with 22 options, none of them matches the designer's plan since they

[32]This action takes place in the role diagram (not shown).

all aim to make the new message internal to one of the existing agents.

2. Link "TAFData" message with the "GUI" agent (receive) (done by the designer).
   The tool identifies several violated constraints related to the new message and the domain specific constraint concerning subscription. It then returns four options, all of which aim, however, to make "TAFData" message sent by the "Alerter" agent, which does not match the designer's plan.

3. Link "TAF Manager" agent with "TAFData" message (send) (done by the designer).
   The tool returns 6 options, one of which matches the designer's plan, resulting in the next 6 steps being done by the tool.

4. Link "ProcessTAF" plan (in "TAF Manager" agent) with "TAFData" message (send) *(done by the tool).*

5. Create a "Use TAFData" plan in "GUI" agent[33] *(done by the tool).*

6. Make "TAFData" message to be a trigger of "Use TAF Data" plan[33] *(done by the tool).*

* 7. Link "SubscribeChange" message with "TAFManager" agent (receive) *(done by the tool).*

* 8. Create "NoteTAFSubscription" plan in "TAFManager" agent *(done by the tool).*

* 9. Make "SubscribeChange" message to be a trigger of "NoteTAFSubscription" plan *(done by the tool).*

*10. Create "SubscriptionsToTAFs" data in "TAFManager" agent (done by the designer).
   The tool returns 4 options, all of which matches the designer's plan. However, each option contains only two steps: (11, 13), (11, 14), (12, 13), (12, 14). The designer chooses one of the options (e.g. 11 and 13) and manually perform the remaining steps

*11. Link "SubscriptionsToTAFs" data with "ProcessTAF" (read) *(done by the tool).*

*12. Link "NoteTAFSubscription" plan with "SubscriptionsToTAFs" data (write) (done by the designer).

[33]This action takes place in the agent overview diagram for agent "GUI" (not shown).

*13. Link "SubscriptionsToTAFs" data "Manage TAF Data" role to (read)[34] *(done by the tool).*

*14. Link "Manage TAF Data" role to "SubscriptionsToTAFs" data (write)[34](done by the designer).

The tool is able to identify and perform 8 actions out of 14 actions in the change plan. There are 8 side-effect actions and five of them have been performed by the tool[35]. Overall, the side-effect actions identified by the tool are very important since they deal with the domain specific requirement related to subscription which is easily missed by the designer (as shown in [Jayatilleke, 2007]).

9.5.7 Summary of all changes

In this section we present and discuss the results. Table 9.1 shows the results of evaluation for all changes. Each change has a row, where the entries marked with numbers show the situation for the *nth* step of the user's plan (D). An entry of the form n_m indicates that the user performed this step[36] and the tool returned n options (i.e. $\mathcal{O} = \{C_1 \ldots C_n\}$), where m of the C_i were compatible with D. There are several cases where the value of m is 0, corresponding to the fact that none of the returned options matched with D. Therefore, the user had to continue performing the next step.

An entry "T" (or "T_s") indicates that an "obvious" (or "side-effect") action is done by the tool, that is, it is part of a selected repair plan from an earlier step. An entry "U" (or "U_s") indicates that the user performs this "obvious" (or "side-effect") action; this occurs in several places where D is non-empty (i.e. there are more changes to be done), but the design is consistent, and in this situation the tool cannot assist the user. The final column gives the value of the metric $M = \mid C \mid / \mid D \mid$. Figure 9.23 shows the portion of changes done by the tool in each change scenario. Overall, for all of the changes the average value of M is approximately 64%, that is, the tool performs on average nearly two-thirds of the actions.

[34]This action takes place in the role diagram (not shown).

[35]In fact, the tool is able to identify all of the side-effect actions since at step 10 it returns four options all of which match some of the remaining side-effect actions. The designer who looks at the different options could be reminded of all the side effects or he/she could select all four options returned by the tool (similar to what could happen at step 6 in change scenario 2).

[36]The form n^s_m indicates this is a side-effect action step.

Change	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	M
1. Wind Speed Alert	9_1	T	T	T	U																60%
2. Variable Threshold	18_1	T	T	T_s	T	26_3	T	T_s	T	3_2	T_s	T	U								69%
3. Volcanic Ash	1_1	T_s	T_s	19_1	T	T	T_s	T	26_1	T	T_s	T	25_0	18_1	T	T	T	T	U	U	65%
4. Logging (alt 1)	24_1	T	T_s	T																	75%
4. Logging (alt 2)	24_1	T	T_s	T	U																60%
5. Multiple TAFs	1_0	2_1	T_s	T_s	27_0	18_1	T	T	T	T	4_1	T_s	T								62%
6. Subscription	22_0	4_0	6_1	T	T	T	T_s	T_s	T_s	4_4^s	T_s	U_s	T_s	U_s							57%

Table 9.1: Summary of evaluation results derived from the six change scenarios

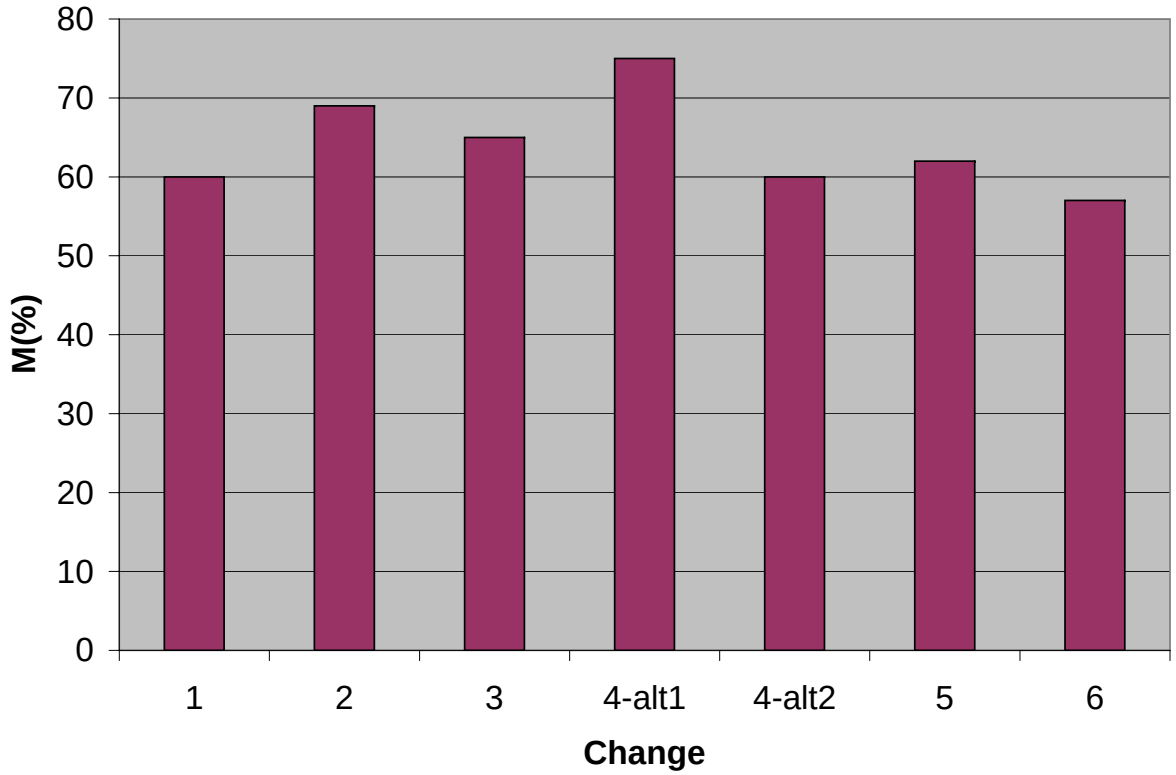


Figure 9.23: The portion of changes done by the tool in each change scenario

This result implies that compared with maintenance without our tool, the user would have to perform roughly triple as many change actions.

Furthermore, for the first five changes, the tool is able to identify and perform all the side-effect actions in the change plan. For change 6, the tool identifies more than half of the side-effect actions. This result implies that without using our tool the user would have to carefully examine the design in order to find and carry out those side-effect actions. In terms of the number of options returned by the tool each time it is used, there are several cases where this number is relatively large (e.g. 26 options after step 6 in change 2). However, due to our mechanism of prioritizing repair plans presented to the user, (i.e. repair plans that affect primary changed entities appear first, as discussed in section 3.3 of chapter 3), the matching option(s) is listed first, which is easily identified by the designer. Nonetheless, it is ideal not to overwhelm the designer with such a large number of options and this issue is part of our future work (discussed in chapter 10).

Overall, these results implies that the tool is effective in terms of helping the designer

identify and perform actions according to a change plan. Especially, for most cases the tool is able to pick out side-effect actions, which tend to be missed by the designer. In the next section, we will investigate the efficiency aspect of our CPA tool.

9.6 Efficiency analysis

In the previous section, we have presented the results of an effectiveness evaluation which shows that our framework is able to produce good recommendations for a real application and change scenarios. However, another key issue that we need to address is the efficiency aspect of our approach.

Generation of repair plans is performed ahead at "compile" time, and is thus not an issue. At runtime, the following steps are performed:

1. Check the design for consistency with respect to provided OCL constraints and a meta-model.
2. Generate repair plan instances for violated constraints.
3. Compute the cost of different repair options.
4. Execute the selected repair plan, where selection is either choosing the cheapest plan or asking the user (if there is more than one cheapest plan).

The fourth step is executing a selected repair plan which is simply a matter of running the plan and performing the changes, and this is quite cheap, since costs have already been computed for the plan and all sub-plans. Therefore, it is not necessary to examine the fourth step in the efficiency analysis. As we discussed in chapter 7, cost calculation (step 3) also involves consistency checking (step 1) and plan instance generation (step 2) because it simulates the change propagation process in doing a look-ahead planning. As a result, the efficiency of the first three steps are considered together in the third step. In addition, the third step is critical since it has the major impact on the performance of our approach. Therefore, the focus of our efficiency analysis is on the cost algorithm. We need to assess how practical the algorithm is, specifically, how well does it scale to larger problems?

In order to investigate this issue we perform a number of experiments where we "stress test" the algorithm in an artificial setting. As discussed in section 7.4 (on page 177), the cost algorithm operates with plan-goal trees, where a goal has as children the plans that can be

used to achieve it. The need to fix a violated constraint is represented as a goal node and different repair instances for fixing it are represented as plan nodes in the plan-goal tree. Furthermore, when calculating the cost of fixing a constraint, the algorithm also considers other constraints in the same repair scope. The need to fix each of those constraints is represented as a subgoal. As a result, two key parameters that we vary are the number of repair plan instances (for one constraint), which corresponds to the *width* of the plan-goal tree; and the overall size of the tree, which we do by varying the number of constraints, and hence the *depth* of the tree. We measure the running time, and how many nodes the algorithm avoided having to explore through pruning. In addition to considering an artificial setting, we also perform some experiments with a non-artificial application.

Our simple artificial setting involves a design that has some number of roles, and some number of agents. All of Prometheus' consistency constraints (as listed in section 4.4) and multiplicity constraints in Prometheus metamodel (section 4.3) are used. However, the only constraint that will be violated in this artificial setting is the one that states that all roles should be associated with an agent: **Context Role inv c** : self.agent→size()≥ 1. This constraint is translated to the following repair plans[37], where *sa* is short for *self.agent*

P1 $c_t(self) \leftarrow$ for each $i \in \{1 \ldots (1 - size(sa))\}$ $!c'_t(self)$

P2 $c'_t(self) : x \in \text{Type}(sa) \wedge x \notin sa \leftarrow$ Add x to sa

P3 $c'_t(self) \leftarrow$ Create x : Type(sa) ; Add x to sa

In order to explore how the algorithm performs as the number of repair plan instances is increased we have a design with a single role and N agents. This gives a single violated constraint to fix, and by increasing N we increase the number of repair options (since there is always a single instance of **P3**, but there are N instances of **P2**, one for each agent).

Figure 9.24 shows the runtime (in milliseconds) for the first experiment[38]. In this experiment pruning made no significant difference, since there is nothing to distinguish between the agents (the results in the graph are from the no-pruning run). Most of the time was taken up with checking for violated constraints in the repair scope (line 20 of figure 7.5 on page 181); for instance, for 160 agents, the total execution time was 1,964ms, of which 1,915ms

[37] The translation is not optimal because it also caters for constraints of the form $size() \geq n$.

[38] All experiments were performed on a laptop running Windows XP and Java v1.5.0_06, with an Intel Centrino 1.73Ghz CPU and 1GB RAM. Times (reported in milliseconds) are an average of 30 runs (we ignored the first run, since it was inconsistent due to JVM startup). For each run we collected the number of goal and plan nodes explored, the total time (broken down into the constraint evaluation time, time taken to update models, and other time), and the number of constraint instances.

was taken in constraint evaluation. In addition, the figure also shows that the algorithms are exponential in the number of violated constraints in a repair scope due to an extensive look-ahead planning as discussed in section 7.4.4.

One technique (proposed by [Egyed, 2006]) which we have not applied, but which we expect to make a big difference to execution time, is to track which entities are used to evaluate each constraint, and then use this information to work out which constraints might be affected by a change to the design, and only re-evaluate these constraints. However, even without this, the algorithm is able to deal with a reasonable number of repair plan instances quite rapidly (just under two seconds for 161 design entities and 1,606 constraints).

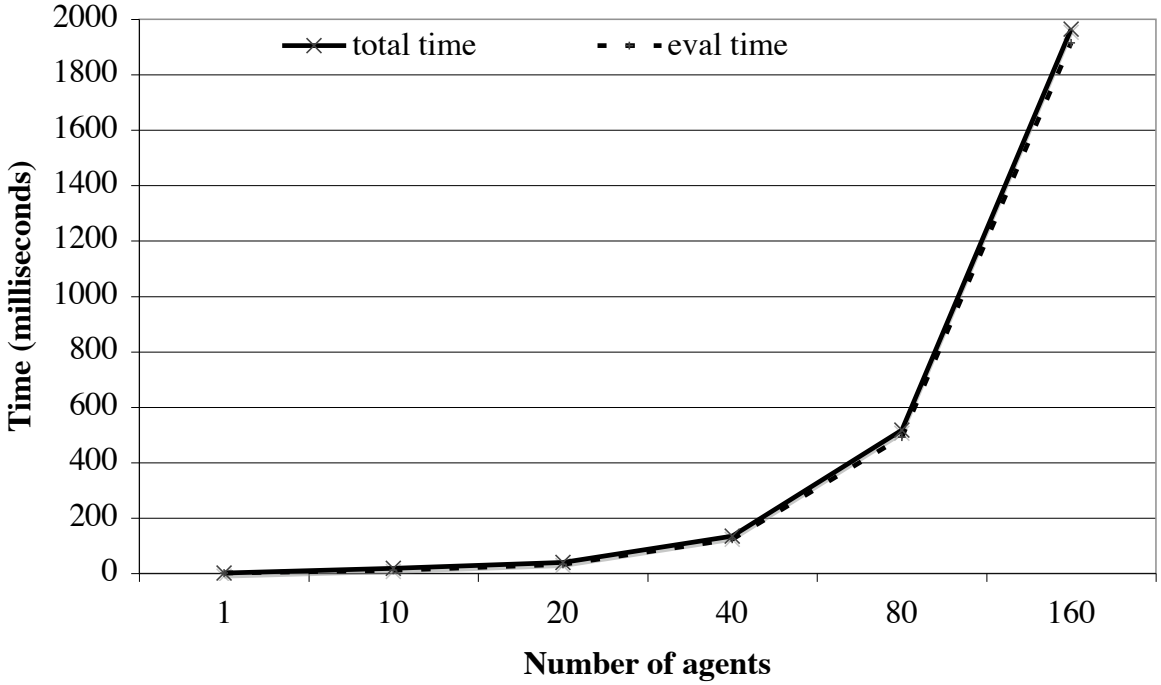


Figure 9.24: Performance of the cost algorithm in the first experiment

We now consider the algorithm's performance as the number of constraints, and consequently the depth of the tree, is increased. We create an artificial situation with N constraints by having N roles and one agent. Since each role has a single violated constraint, this gives N constraints, and consequently a tree of depth N.

In this case pruning made a significant difference: for $N = 8$ without pruning the algorithm considered 10,590 goal nodes and 31,736 plan nodes taking a total of 21,594 milliseconds, whereas with pruning the algorithm considered 1,673 goal nodes and 3,753 plan nodes taking

1,360 milliseconds. On the other hand, the heuristic of sorting plans by their basic cost (line 5 of figure 7.6 on page 182) made no difference. As figure 9.25 shows[39], the algorithm (with pruning) performs well for $N = 8$. In this experiment the evaluation time was a smaller component of the total time.

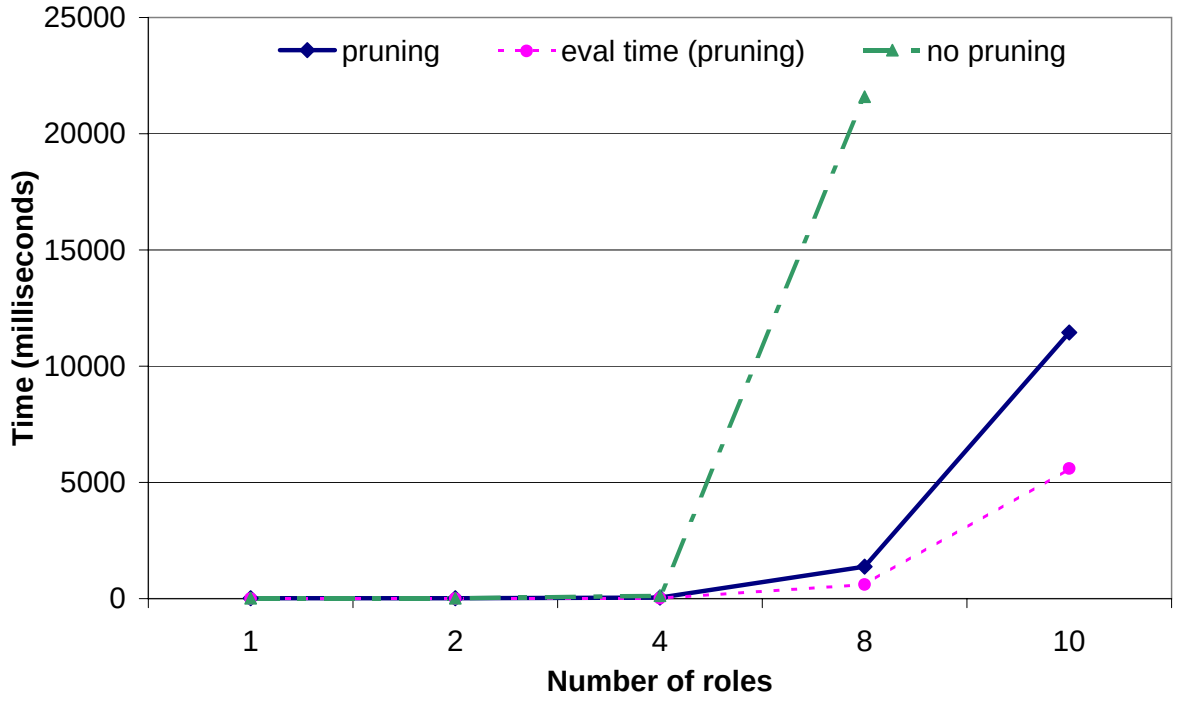


Figure 9.25: Performance of the cost algorithm in the second experiment

Finally, in order to assess the performance of the algorithm in a non-artificial situation, we conducted experiments on the design of the weather alerting system that we used to assess the effectiveness of our approach[40]. We report here the six change scenarios that are discussed in section 9.5. The existing design model contains 85 elements, and 64 consistency constraints and multiplicity constraints are considered. With regard to the no-pruning case, the tool does not terminate as the algorithm fell into a cycle[41] . We report here the performance

[39]For $N = 10$ the no pruning case ran out of memory.

[40]All experiments were performed on a laptop running Windows XP and Java v1.5.0_06, with an Intel Centrino 1.73Ghz CPU and 1GB RAM. Times (reported in milliseconds) are an average of 5 runs (we ignored the first run, since it was inconsistent due to JVM startup). For each run we collected the number of goal and plan nodes explored, the total time (broken down into the constraint evaluation time, time taken to update models, and other time), and the number of constraint instances.

[41]The version that fell into the loop was the algorithm with the loop detection turned off, and the version with loop detection enabled successfully avoided an infinite loop.

Change	Invoke	Constraint Instances	Goal Nodes	Plan Nodes	Total Time (ms)	Eval Time (ms)
1. Wind Speed Alert	1	306	475	1034	860	783
2. Variable Threshold	1	307	674	1093	1297	1141
	2	320	1143	1808	1578	1391
	3	320	117	185	250	218
3. Volcanic Ash	1	322	61	91	265	233
	2	331	2848	4830	4984	4467
	3	344	2161	3609	2437	1906
	4	346	3539	5602	5234	4739
	5	346	4713	7755	52203	46016
4. Logging[42]	1	306	2794	4355	4922	4249
3. Multiple TAFs	1	304	21	34	93	47
	2	304	79	118	359	312
	3	346	3593	5698	5359	4907
	4	346	5665	9458	7984	6537
	5	317	111	179	313	266
6. Subscription	1	306	2764	4382	3859	3545
	2	311	2884	4795	4438	4001
	3	311	3514	6512	6203	5485
	4	335	2723	4369	5609	3630

Table 9.2: Efficiency results from the six change scenarios

of the algorithm that has pruning and the plan sorting heuristic. Table 9.2 shows how the algorithm performs each time the tool is invoked (column 2) during every change scenario (column 1) in terms of the number of constraint instances (column 3), goal nodes (column 4), and plan nodes (column 5), the total time (column 6), and evaluation time (column 7). It is noted that "invocations" of the tool correspond to the entries in Table 9.1 that have numbers. For instance, in the second change scenario (Variable Threshold) the tool is invoked three times at steps 1, 6, and 10.

The results implies that, despite a worse case exponential complexity, the algorithm is practical for small to medium designs. Note that there are still a number of techniques for improving the algorithm's efficiency which we have not yet implemented (which will be discussed in chapter 10).

9.7 Discussion

The evaluation demonstrated that the approach is effective if a reasonable amount of primary changes is provided. One issue that arises in the proposed approach relates to the use of inconsistency as a driver for change. As seen in the evaluation, not all changes result in inconsistency, and in these cases the approach will not be able to completely identify the

desired secondary changes. An opposite issue is that, as argued by [Fickas et al., 1997], not all inconsistency should be fixed; this is easy to deal with by simply allowing certain constraint types or instances to be marked as "to be ignored".

In some cases there may be a large number of repair options returned by the tool, which makes it hard for the user to select which one to use. In practice this can be dealt with by ignoring the tool's list of options and performing further changes (which often provides the tool with information that enables it to return fewer options). A better approach which needs to be investigated is reducing the number of options by "staging" questions. Suppose we need to link a percept with a plan and with an agent, then instead of presenting a set of options, where each option specifies both a plan and an agent (which gives a cross product), specify first the choice of agent, and then based on that choice ask for a choice of (relevant) plan.

Overall, our conclusion is positive since the evaluation shows that the approach is able to (on average) perform (approximately) more than two-thirds of the actions in maintenance plans, across a number of changes motivated by experience with a real application.

However, there are several issues and limitations in our evaluation study. Firstly, our goal was to maximize the internal validity of our approach. We have used a range of change scenarios in the context of the designer using Prometheus to evolve an agent-oriented design. Since our approach performed well for all these change scenarios, we believe that there is little threat to the internal validity of the effectiveness evaluation outcome. However, we were not able to directly observe the designers in their use of our approach. As a result, future work is needed to conduct the usability study.

Another issue is related to the generalization of the results of our study. There are several questions related to this issue. Firstly, can we generalize the approach to other software engineering methodologies, design models or even source code? Secondly, can we generalize our results to other software systems and change scenarios in practice?

We have tried to address the first question by showing the applicability of our approach to a representative of an object-oriented methodology (UML) and an agent-oriented methodology (Prometheus). However, it was not possible to perform an extensive study on the application to UML, which is a target for our future work. We have also performed a preliminary investigation (not reported here) on how our approach can be extended to support design and source code change propagation and found that it is feasible provided that the

gap between design and code is relatively small. For instance, UML 2.0 (object-oriented) [Object Management Group, 2004] and CAFnE[43] (agent-oriented) [Jayatilleke et al., 2005] have such a potential.

In order to answer the second question, we performed a case study based on a real application. In addition, the set of changes are motivated by real change request and cover most of the change types. However, there are several threats to the external validity of our study. Firstly, the changes that we investigated in our evaluation may not be representative of all changes in reality. In addition, we need to test the approach with different application types and sizes. However, the efficiency evaluation has shown that our cost algorithms are practical for small to medium realistic examples. Future work is needed to make the approach more scalable, and applicable to larger designs.

Finally, how well our approach performs depends on the number of constraints and their quality. A large number of constraints may make the tool slow to respond because more time is spent on checking constraints and working out different plans for fixing violated constraints. By the same token, having more constraints tends to make the tool more helpful in terms of identifying necessary secondary changes that are possibly missed by the designer. With regard to the quality of constraints, our approach assumes that constraints are correctly specified and that they come from a reasonable source (as discussed in chapter 3).

9.8 Chapter summary

In this chapter, we have reported on an evaluation which was performed to assess the effectiveness and efficiency aspects of our framework and its prototype implementation, i.e. the Change Propagation Assistant (CPA) tool. We have first identified the key issues that we faced when carrying out the evaluation. These issues include choosing the methodology to which our framework is applied, selecting the application to test with our approach, identifying different change scenarios which may occur to the evaluation application, selecting primary changes, and determining basic costs. We have discussed and proposed our solutions to those issues. Furthermore, one of the major obstacles during the evaluation is the lack of a user study due to limited time and resources. The solution that we have proposed to deal with this issue is defining an abstract user behaviour in maintaining/evolving an existing

[43]CAFnE is an extension of the Prometheus Design Tool that allows more design details to be specified and produces fully executable code.

design. With respect to this definition, the change propagation process is driven by a user's change plan. We then simulated a real user by performing steps in this plan and assessed the responses from the CPA tool.

In addition, we have discussed an experimental process and several metrics that we have used to assess our approach in general and our CPA tool in particular. An important metric is to measure what proportion of the actions in the change were identified and performed by the CPA. We have also distinguished "obvious" change actions from "side-effect" actions in order to further assess the usefulness of our CPA tool.

Finally, we have described how the experimental process and those metrics are applied in a case study involving the maintenance of the Prometheus design of a weather alerting system. We have discussed and analysed the results of six different change scenarios occurring in that application. Furthermore, an efficiency analysis based on artificial settings and the evaluation application have also been reported.

This chapter completes the description of the research carried out in this thesis. In the next chapter, we will highlight the main problems that we wanted to solve and summarise our key contributions in addressing those problems. We will also discuss some major limitations in our approach and then propose future work to deal with them.

Chapter 10

Conclusions and Future Work

Software maintenance and evolution are inevitable activities since almost all software that is useful and successful stimulates user-generated requests for change and improvements. One of the most critical problems in software maintenance and evolution is to maintain consistency between software artefacts by propagating changes correctly (i.e. *change propagation*). Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software engineering.

As agent-oriented approaches represent an emerging paradigm in software engineering, an increasing number of agent applications have been developed in the past few years. Similar to conventional software systems, agent-based applications also evolve to meet ever-changing user requirements and environment changes. Therefore, it is important to investigate solutions that help improve the practice of maintaining and evolving agent systems.

However, there has been very little work that we are aware of on software maintenance in agent-oriented software engineering. Our work aims to fill that gap, as well as apply agent technology to the problem of software maintenance in a broader context. This chapter concludes the work that has been carried out, puts it back into context and provides some pointers to possible future lines of research.

10.1 Summary of contributions

Our major objective was to provide (semi-)automated support for change propagation in and between design models. We followed an inconsistency-based approach in which change propagation is carried out by fixing inconsistencies in a design. In other words, we propagate

changes by finding places in a design where the desired consistency relationships are violated, and fixing them until no inconsistency is left in the design. The following research questions were raised in chapter 1:

1. What type of consistency relationships can be derived from design artefacts to support change propagation and how can they be identified and represented?

2. How to effectively represent and implement a mechanism for propagating changes by fixing those consistency relationships when they are violated?

3. (a) What is an appropriate representation for capturing different options of repairing an inconsistency? (b) What kinds of automation can be provided to generate such repair options?

4. How to select between different applicable repair options to fix a given consistency violation?

5. What type of tool support can be given to designers to assist in the process of understanding and modifying an existing agent system?

We developed an agent-based change propagation framework, addressing each of those research questions as summarised below.

We view the central asset of the design as being a model, a high-level description of the system under development. Design artefacts correspond to different views of the model which describe various aspects of the system (chapter 3). From our perspective, a model must be an integrated, consistent and coherent unit, which requires each view of a model to be both syntactically and semantically consistent. We used a metamodel (as a form of the model's abstract syntax) to define syntactic consistency conditions that each model's view must obey to guarantee that the overall model is well-formed (*research question 1*). We used the Unified Modelling Language (UML) to specify metamodels, although other modelling languages such as the Meta-Object Facility (MOF) can also be used.

In addition, we employed a set of constraints to define consistency conditions that cannot be expressed using a metamodel (*research question 1*). Such constraints are used to describe syntactic relationships between metamodel elements. They may also be used to prescribe coherence relationships (semantically consistent) between different views of a model, i.e. intra-model or horizontal consistency. Although we did not fully investigate this, our

approach is also applicable to constraints that express consistency between different models, i.e. inter-model or vertical consistency, provided that those models are defined based on the same metamodel. Constraints are also used to describe domain-specific requirements, and impose best practices or industry standards on designers. We used the Object Constraint Language (OCL), which is also part of the UML standards, to specify constraints. In order to illustrate how consistency relationships can be identified and represented following the above approach, we developed a metamodel and a set of consistency constraints for the Prometheus methodology (chapter 4).

The novel aspect of our framework is the underlying change propagation mechanism which uses agent technology (*research question 2*). Specifically, the change propagation engine in our framework is represented and implemented using the well-known Belief-Desire-Intention (BDI) agent architecture (chapter 3). Although we did not use the full capabilities of the BDI model, the adoption makes use of various properties of BDI agents to gain advantages over more traditional approaches to change propagation. BDI agents operate in an event-triggered manner, where events trigger plans, which in turn can create new events resulting in further plans being triggered. We exploit this to reflect the cascading nature of change propagation, where performing an action to fix an inconsistency can cause further inconsistencies (*side-effects*) which require further actions. In addition, an event can have multiple plans that it can trigger, with plan selection being made at run-time. This allows us to represent multiple ways of resolving a given inconsistency as separate plans, with the choice between them determined by plans' context condition, corresponding to available information such as the cause of inconsistencies, design heuristics and (possibly) human intervention. Moreover, another advantage is that new (or alternative) ways to resolve an inconsistency can be added via additional plans without changing the existing structure.

As a result, we represented constraint violation as events and the strategies of propagating changes to resolve model inconsistencies (caused by constraint violation) are represented as (repair) plans. We proposed a formal abstract syntax for repair plans based on AgentSpeak(L) (*research question 3a*). This allows for repair plans to be abstractly represented as a way to reasonably enumerate the otherwise large number of concrete ways of fixing inconsistencies.

Nonetheless, the number of repair plan types can be very large, especially for designs that are restricted by a substantial number of consistency constraints. In these cases, hand-

crafting sound and complete repair plans for all constraints becomes a difficult and labour intensive task. To address this issue, we proposed and implemented the repair plan generator (chapter 6), an important and novel component in our change propagation framework (*research question 3b*). The key idea of this generator is automatically producing repair plans for a given OCL constraint. The repair plan generator is driven by the set of generation rules for common OCL expressions including navigations, attributes, set operations, and boolean connectives. Those rules were carefully designed to ensure that the generated plans are of high quality and not faulty. In fact, we formally proved that the repair plans generated following our rules are correct, minimal and complete. The repair plan generator was developed and tested with a subset of UML well-formedness constraints and a set of OCL consistency constraints that we developed for the Prometheus methodology.

Our framework has a plan library that contains repair plan types, which are all created by the repair plan generator. However, we also allow the designer to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed. In such cases, the user must be responsible for checking the correctness and completeness of their plans.

Repair plan types are instantiated at run time to become plan instances. For a given constraint violation, there can be different applicable (repair) plan instances that are able to fix it. This leads to the general problem of (applicable) BDI plan selection in the context of change propagation: how to select between different applicable repair plans to fix a given constraint violation (*research question 4*). We investigated this issue and observed that the decisions upon which repair plans are chosen can depend on various factors which require extra knowledge provided by the software designer. Therefore, we argued that it is not feasible to have a completely automated mechanism for selecting repair options. As a result, our solution is to develop a semi-automated mechanism for repair option selection: options that are considered infeasible and costly are automatically filtered out, and the user is presented with a set of "quality" repair options for selection.

Those ideas were developed in terms of a (repair) plan cost calculation component within our framework (chapter 7). The mechanism of plan selection is based on a notion of repair plan cost. In addition, it provides a simple mechanism for the user to adjust the change propagation process in terms of assigning costs for basic repair actions. Moreover, our notion of cost considers the side-effects of a repair plan as a component of the repair plan's cost. In

this context, positive side-effects decrease the cost of a repair plan whilst negative side-effects augment its cost.

We provided formal definitions for the cost of actions, plans, constraints and (sub)goals. In addition, we were able to prove an important property derived from those definitions, which indicates that the total cost of fixing a set of constraints in a repair scope does not depend on the order in which constraints are fixed. This has enabled us to derive algorithms that calculate the cost of fixing violated constraints. We presented and discussed two versions of the cost algorithms: one involving an exhaustive search and the other with pruning capabilities which improve the performance of the algorithms. The cost algorithms were developed to find cheapest repair options and propose them to the user for final selection.

In order to demonstrate the applicability and practicality of our approach, we developed a proof-of-concept tool support in the form of a prototype tool called Change Propagation Assistant (CPA) (*research question 5*, chapter 8). The CPA tool was integrated with the Prometheus Design Tool (PDT) to support software designers in maintaining and evolving Prometheus designs. Although the implementation is specific to the Prometheus methodology, our framework is generic and can be applied to a range of design types. In fact, the implementation architecture implies that Change Propagation Assistant can not only be integrated with PDT, but also be modified to be plugged into other modelling environments. In addition, we also showed that our approach is applicable not only to agent-oriented models (e.g. Prometheus models) but also to object-oriented design models such as UML models (chapter 5).

Finally, we used the CPA tool to perform an empirical evaluation to assess the efficiency and effectiveness of our change propagation framework (chapter 9). The evaluation's results demonstrated that the approach is effective given that a reasonable amount of primary changes are provided. In terms of efficiency, the evaluation showed that checking for violated constraints takes up most of the execution time (which can be improved by applying more advanced techniques), and that the algorithms are practical for small to medium realistic examples. The evaluation's results also lead us to some potential future work that is discussed in the next section.

10.2 Future work

Our approach can be further improved by addressing some of the limitations where further research and extensions to the change propagation framework are required.

Our repair plan generator has rules that cover a subset of OCL expressions. Although those rules are currently sufficient to deal with consistency constraints defined in Prometheus, we are aware that other OCL expressions (that we have not covered) may be needed. For instance, *let* and *if-then-else* expressions, the *iterate* operation, and collection types other than *Sets* like *Sequences* and *Ordered Set* are not yet addressed. Therefore, potential future work may involve an extension to the repair plan generator to fully support all forms of OCL expressions. In addition, the main focus of our work is on syntactic well-formedness and coherence between models. However, in practice, consistency can involve not just syntactic, but also semantic constraints and domain models. Further, it is not clear to what extent OCL is suitable to capture such constraints. Therefore, a topic for future work is to explore such constraints and investigate how to extend our framework to support them.

Our algorithms for calculating repair plan costs require an extensive look-ahead search which is expensive. In fact, our complexity analysis and efficiency evaluation showed that our cost algorithms are only practical for small to medium realistic examples. As a result, future work is needed to make the approach more scalable, and applicable to larger designs. This may include an investigation of the interaction between constraints in order to limit the number of plans to be explored and to allow for pruning more quickly. In addition, the evaluation showed that checking for violated constraints takes up most of the execution time. Therefore, future work may involve applying more advanced constraint checking techniques to improve the overall performance. For example, an approach recently proposed by Egyed [2006] uses a form of model profiling to quickly, correctly, and automatically decide when to evaluate consistency constraints. We expect that employing such techniques would make a substantial difference to the execution time of our cost algorithms.

The evaluation assessing the effectiveness of our CPA tool showed that in some cases the tool proposed a large number of repair options, which makes it difficult for the user to decide which one to use. This issue leads to a topic for future work concerning a method to reduce the number of repair options presented to the user. An approach that we mentioned earlier is "lazy" selection (or "staging" questions). For example, assume that we need to link a percept with a role and with an agent, then instead of presenting a set of options, where each option

specifies both a role and an agent (which gives a cross product), specify first the choice of role, and then, based on that choice, ask for a choice of (relevant) agent.

We would also like to perform a more extensive study on the applicability of our approach to other design types such as UML models. This may result in an implementation of a CPA-like tool that can be integrated with existing UML modelling tool (e.g. ArgoUML[1]). Since the UML metamodel is already available [Object Management Group, 2005], we will not need to develop a metamodel for UML models. Consistency constraints in the form of OCL are also included as part of the UML standard [Object Management Group, 2005], which can be input to our framework. In addition, an interesting topic for future work is to apply our approach to change propagation between design models and source code.

Finally, there are several potential extensions to our evaluation for assessing the effectiveness of our approach. Firstly, assigning costs to basic repair actions (e.g. creation, connection, disconnection, etc.) is a means for the user to adjust the change propagation process. The results of our evaluation also indicate several places where the outcome may be sensitive to the basic costs. As a topic for future work, we want to perform a more thorough exploration of the effects of varying the basic costs. Moreover, we have used a cost algorithm to estimate and drive the inconsistency repair process in such a way that we assume the cheapest repair plans are preferable from the perspectives of the user. Although in the case study reported in chapter 9 the cheapest solution also turns out to be the one that is intuitively the most appropriate, the cheapest cost heuristic may not always lead to the best way to resolve inconsistencies. As part of future work, we would like to perform more detailed user experiments to have more understanding on this issue. In addition, we would also like to test our approach with a range of different realistic applications and change scenarios. Furthermore, we want to improve the usability of the CPA tool and perform an evaluation study which involves real software designers using our tool. This would give us much more information on how helpful the tool is in practice. Since the tool is still a prototype that is not ready for industry use, one area of future work is to develop an industry-grade tool. Nonetheless, although the results obtained from the evaluation we conducted are relatively preliminary and necessarily limited, they are quite encouraging and serve as concrete indication that the approach developed is promising.

[1] http://argouml.tigris.org

Appendix A

Proof

In this appendix we prove that theorem 1[1] (in section 6.6 on page 155) holds for all basic OCL constraints that we cover in section 6.4, with respect to the translation schemas for making a constraint true (i.e. c_t) or false (i.e. c_f). Our proof uses induction over the structure of a constraint.

The proofs follow the same style which starts by examining the repair plan types derived in each rule. We need to show that $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing constraint c. First, we examine a set of possible action sequences AS obtained from instantiating and resolving the plan types generated for a given constraint. We then identify all possible minimal ways of fixing the violated constraint. Finally, we argue that the set of action sequences resulting from the generated repair plans for each constraint are precisely the possible minimal action sequences that fix the constraint.

The proofs are presented following the structure of the presentation of the repair plan generation rules in section 6.4 (on page 132), i.e. rules are organised into various groups including navigation, constraints on attributes, constraints on Boolean-valued set expressions, constraints on non-Boolean-valued set expressions, Boolean connectives, and addition and deletion involving derived sets.

[1]"For any given OCL constraint C which is satisfiable, the set of repair plans $\mathcal{R}(C)$ produced by the repair plan generator is correct and complete. That is, it generates a representative permutation for each correct and minimal action sequence, and does not generate any incorrect action sequences, and all generated action sequences are minimal."

A.1 Proofs for generated repair plans for making a constraint true, i.e. $\mathcal{R}(c_t)$

A.1.1 Navigation

In this section, we provide proofs for basic constraints concerning navigations leading to a single entity.

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend = x}) =$

$\{c_t$: E.aend = null $\leftarrow$ Connect E and x (w.r.t *aend*),

c_t : E.aend = y[2] $\leftarrow$ Disconnect E and y (w.r.t *aend*); Connect E and x (w.r.t. *aend*)$\}$

Proof: This rule assumes that $E.aend$ leads to a single entity, which can result in two cases.

Case 1: E is not connected to any entity with regard to association end *aend*

In this case, $E.aend = null$ and consequently from the plan definitions only the first plan is applicable. Therefore, AS has a single *as* which contains only one action that connects E to x (w.r.t. *aend*).

Constraint c is violated if and only if E is not connected to x with regard to *aend*, i.e. $E.aend \neq x$. Also, we know that E is not connected to any entity with regard to *aend*. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E.aend = x$ which necessarily involves connecting E and x (w.r.t. *aend*). As can be seen from the plan generated by $\mathcal{R}$ in this case, the generated plan exactly follows this pattern, and thus the action sequence resulting from the plan generated by $\mathcal{R}(c_t)$ correctly and minimally repairs c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must connect E and x (w.r.t. *aend*), and this is exactly what is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case[3].

Case 2: E is connected to entity y with regard to association end *aend*

In this case, $E.aend = y$ and consequently from the plan definitions only the second plan is applicable. Therefore, AS has a single *as* which contains only two actions that disconnect E and y (w.r.t. *aend*), and connect E to x (w.r.t. *aend*).

[2] In this case, $E.aend$ is implicitly not *null*, i.e. $y \neq null$.

[3] In fact, there is only a single action sequence, which contains only a single action, so there are not really any permutations, and the resulting set of actions is complete in the stronger sense of containing all action sequences rather than containing representative permutations.

Constraint c is violated if and only if E is not connected to x with regard to $aend$, i.e. $E.aend \neq x$. Also, we know that E is connected to y with regard to $aend$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E.aend = x$ which necessarily involves first disconnecting E from y since $E.aend1$ can only connect to one entity, and then connecting E and x (w.r.t. $aend$). As can be seen from the plan generated by $\mathcal{R}$ in this case, the generated plan exactly follows this pattern, and thus the action sequences resulting from the plan generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend1.aend2 = x}) =$

$\{c_t$: E.aend1 = y $\leftarrow$ $!(y.aend2 = x)_t$,

c_t : z.aend2 = x $\leftarrow$ $!(E.aend1 = z)_t$,

c_t : z $\in$ Type(E.aend1) $\wedge$ z.aend2 $\neq$ x $\leftarrow$ $!(E.aend1 = z)_t$;$!(z.aend2 = x)_t$,

$c_t \leftarrow$ Create z : Type(E.aend1); $!(E.aend1 = z)_t$; $!(z.aend2 = x)_t$ $\}$

Proof: This rule assumes that $E.aend1$ leads to a single entity and so does $E.aend1.aend2$, which results in the following cases. It is noted that the fourth plan type is always applicable since it does not have any context condition.

Case 1: There is no entity connected to x with regard to $aend2$

In this case, there does not exist any entity z such that $z.aend2 = x$ and thus the second plan type is *not* applicable. By contrast, the third plan type is applicable and generates plan instances, each of which corresponds to an entity z that has the same type as $E.aend1$. We now consider the following sub-cases.

Case 1.1: E is connected to an entity y (w.r.t. $aend1$)

In this case, $E.aend1 = y$ and consequently the first plan is applicable. Hence, we now have three applicable repair plan types. From the definitions of these plans, any action sequence as in AS contains actions that either make $y.aend2 = x$ true (as in the first plan), or make both $E.aend1 = z$ and $z.aend2 = x$ true, where z is either an existing entity that has the same type as y (as in the third plan) or z is a newly created entity (as in the fourth plan).

Making $y.aend2 = x$ true can yield further action sequences as previously discussed

in the case of making a constraint in the form of $E.aend = x$ true. For instance, if y is not currently connected to any entity (w.r.t. $aend2$), then making $y.aend2 = x$ true is achieved by connecting y and x (w.r.t. $aend2$). Otherwise, we need to disconnect y with its connected entity (w.r.t. $aend2$) first, and then connect y and x (w.r.t. $aend2$). Similarly, making $E.aend1 = z$ (or $z.aend2 = x$) can yield further action sequences. We have already proved that such action sequences are complete, correct and minimal (refer to the proof for $c \stackrel{\text{def}}{=} E.aend = x$ on page 261). We use this result to prove by induction that repair plans posting the events of making $y.aend2 = x$ true, or making $E.aend1 = z$ and $z.aend2 = x$ true also yield complete, correct, and minimal action sequences.

Constraint c is violated if and only if E is not connected to x with respect to $aend1.aend2$, i.e. $E.aend1.aend2 \neq x$. Also, we know that there is no entity connected to x with regard to $aend2$, i.e. we cannot find any existing entity z such that $z.aend2 = x$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E.aend1.aend2 = x$ which necessarily involves ensuring that $E.aend1 = z$ for some z and that $z.aend2 = x$. There are three specific possible patterns of action sequences depending on the choice of z: z can be a newly created entity, or it can be an existing entity. In the latter case there are two distinct situations: z can be the (unique) entity for which $E.aend1 = z$ currently holds, or it can be another existing entity. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow these possibilities, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow one of three patterns, each of which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case.

Case 1.2: E is not connected to an entity (w.r.t. $aend1$)

In this case, $E.aend1 = null$ and consequently the first plan type is not applicable. As a result, only the third and fourth plan types are applicable. Therefore, the only minimal way to make c true is making $E.aend1 = z$ and $z.aend2 = x$ true, where z is either an existing entity that has the same type as y or a newly created entity. We use a similar proof to show that AS is complete, correct and minimal.

Case 2: There are entities connected to x with regard to $aend2$

We also consider the two cases as above and use a similar proof. It is, however, noted

that the second plan type is now applicable and generates plan instances, each of which corresponds to an entity z such that $z.aend2 = x$. Therefore, AS also contains action sequences that make $E.aend1 = z$ where $z.aend2 = x$ already holds. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.aend1 = E2.aend2}) =$

$\{c_t$: E2.aend2 = x $\leftarrow$!$(E1.aend1 = x)_t$,

c_t : E1.aend1 = x $\leftarrow$!$(E2.aend2 = x)_t$,

c_t : x $\in$ Type(E1.aend1) $\wedge$ x $\neq$ E1.aend1 $\wedge$ x $\neq$ E2.aend2

$\leftarrow$!$(E1.aend1 = x)_t$; !$(E2.aend2 = x)_t$,

$c_t \leftarrow$ Create x : Type(E1.aend1); !$(E1.aend1 = x)_t$; !$(E2.aend2 = x)_t\}$

Proof: From the plan definitions, any action sequence *as* in AS contains actions that make $E1.aend1 = x$ and $E2.aend2 = x$ true, where x can be either a new entity (as in the fourth plan) or an existing entity[4]. For the latter case, x can be either the current value of $E2.aend2$ (as in the first plan), or the current value of $E1.aend1$ (as in the second plan) or an existing entity that has the same type of $E1.aend1$[5]. Making $E1.aend1 = x$ ($E2.aend2 = x$) true can yield further action sequences as previously discussed for making constraints in the form of $E.aend = x$ true. We have also proved that such action sequences are complete, correct and minimal. We use this result to prove by induction that repair plans posting the events of making $E1.aend1 = x$ ($E2.aend2 = x$) true also yield complete, correct, and minimal action sequences.

Constraint c is violated if and only if $E1.aend1$ does not lead to the same entity as $E2.aend2$ does, i.e. $E1.aend1 \neq E2.aend2$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E1.aend1 = E2.aend2$ which necessarily involves ensuring that $E1.aend1 = x$ and $E2.aend2 = x$ for some x. There are four specific possible patterns of action sequences depending on the choice of x : x can be a newly created entity, or it can be an existing entity. In the latter case there are three distinct situations: x can be the (unique) entity for which either $E1.aend1 = x$ or $E2.aend2 = x$ currently holds, or it can be another existing entity. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow the four patterns, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we

[4]We assume that a solution that sets both $E1.aend1$ and $E2.aend2$ to *null* is not desirable, but that if it is acceptable, then it can be added as an extra repair plan.

[5]Here, we assume that $E1.aend1$ and $E2.aend2$ leads to entities that have the same type.

have argued that any action sequence that correctly and minimally repairs c must follow these patterns, each of which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

A.1.2 Constraints on attributes

In this section, we present proofs for basic constraints concerning attributes of model entities.

- $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr = val}) =$

  $\{c_t \leftarrow$ Change *attr* of E to *val*$\}$

Proof: This plan has no context condition, and thus it corresponds to exactly one plan instance. Therefore, AS contains single action sequence *as* that has one action which changes *attr* of E to *val*.

Constraint c is violated if and only if the value of attribute *attr* of entity E is not equal to *val*, i.e. $E.attr \neq val$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E.attr = val$ which necessarily involves changing the value of *attr* to *val*. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

- $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.attr > val})$[6] =

  $\{c_t :$ Ask(userVal, "> val") $\wedge$ userVal > val $\leftarrow$ Change *attr* of E to *userVal*$\}$

Proof: This plan has a context condition, which contains a function asking the user to provide a value (i.e. *userVal*) which is greater than *val*. If the value provided by the user satisfies this condition (i.e. $userVal > val$) then there is an action sequence *as* in AS that changes the value of *attr* of E to *userVal*. Otherwise, i.e. $userVal \leq val$, the plan is not applicable and nothing is done, resulting in the constraint still being violated. We, however, know that $\mathcal{R}_t(c)$ also contains two other plans $\{fixC_t : c \leftarrow true, fixC_t : \neg\ c \leftarrow c_t;\ fixC_t\}$. This means that plan $fixC_t$ is called recursively until c becomes true and thus the plan c_t is

[6] A similar proof can be used for c $\stackrel{\text{def}}{=}$ E.attr < val

called again until the user provides a value *userVal* that satisfies $userVal > val$. Therefore, AS ultimately contains an action sequence as that changes the value of $attr$ of E to $userVal$.

Constraint c is violated if and only if the value of attribute $attr$ of entity E is less than or equal to val, i.e. $E.attr \leq val$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E.attr > val$ which necessarily involves changing the value of $attr$ to a value that is greater than val. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

- $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E1.attr1 = E2.attr2})$[7] =

  $\{c_t$: E1.attr1 = val $\leftarrow !(E2.attr2 = val)_t$,

  c_t : E2.attr2 = val $\leftarrow !(E1.attr1 = val)_t$,

  c_t : Ask(userVal, "$\neq$ E1.attr1 and $\neq$ E2.attr2") $\wedge$ userVal $\neq$ E1.attr1 $\wedge$ userVal $\neq$ E2.attr2
  $\leftarrow !(E1.attr1 = userVal)_t$; $!(E2.attr2 = userVal)_t\}$

Proof: The above plan types generate action sequences that either make $E2.attr2 = val$ true or make $E1.attr1 = val$ true or make both $E1.attr1 = userVal$ and $E2.attr2 = userVal$ true, where *userVal* is a value provided by the user which is different from both $E1.attr1$ and $E2.attr2$. Making $E2.attr2 = val$ (or $E2.attr1 = val$) can yield further action sequences as previously discussed for making a constraint in the form of $E.aend = x$ true. We have also proved that such action sequences are complete, correct and minimal. We use this result to prove by induction that repair plans posting the events of making $E2.attr2 = val$ (or $E2.attr1 = val$) true also yield complete, correct, and minimal action sequences.

Constraint c is violated if and only if the value of $attr1$ of $E1$ is not equal to the value of $attr2$ of $E2$, i.e. $E1.attr1 \neq E2.attr2$. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that $E1.attr1 = E2.attr2$ which necessarily involves ensuring that $E1.attr1 = val$ and $E2.attr2 = val$. There are three specific possible patterns of action sequences depending on the choice of val: it is either the current value of $E1.attr1$ or the current value of $E2.attr2$ or a new value provided by the user (i.e. $userVal$). As can

[7] A similar proof can be used for c $\stackrel{\text{def}}{=}$ E1.attr1 > E2.attr2

be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow these three possible patterns, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow one of three patterns, each of which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

A.1.3 Constraints on Boolean-valued set expressions

In this section, we present proofs for basic constraints concerning set expressions that return Boolean values. The proofs generally follow an induction approach in which we assume that the set of generated repair plans for adding (or removing) an entity to (or from) a derived set is correct and complete[8].

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)}) = \{c_t \leftarrow$!+(x, SE)$\}$

Proof: From the above plan type definition, any action sequence as in AS contains actions that add x to SE.

Constraint c is violated if and only if x does not belong to SE. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that *SE→includes(x)* which necessarily involves adding x to SE. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}includesAll(SE')}) =$
$\{c_t \leftarrow$ for each x in (SE' - SE) !c'_t(x)
c'_t(x) $\leftarrow$!-(x, SE'),
c'_t(x) $\leftarrow$!+(x, SE)$\}$

Proof: Let $\{x_1, \ldots, x_n\} = SE' - SE$ (where we know that $n > 0$, else the constraint is already true), and, because it is a set, we also know that $x_i \neq x_j$ for $i \neq j$.

[8] We prove this assumption in section A.3.

From the above plan type definitions, any action sequence *as* in AS contains actions that for each x_i either remove it from SE' or add it to SE.

The above constraint can be written as:

$$c \stackrel{\text{def}}{=} \forall\, x \in SE' \bullet x \in SE$$

Assume that c is violated, i.e. $\neg$ c is true, expressed as follows.

$$\begin{aligned} \neg\, c &\stackrel{\text{def}}{=} \neg\, (\forall\, x \in SE' \bullet x \in SE) \\ &\stackrel{\text{def}}{=} \exists\, x \in SE' \bullet x \notin SE \end{aligned}$$

Now, $SE' - SE$ is the set of x_i for which the condition $x_i \in SE' \wedge x_i \notin SE$ holds. Therefore, constraint c is violated if and only if there is an x_i which is in SE' and not in SE. Hence to fix c (minimally) must ensure that no such x_i exists. We do this by finding all x_i for which the condition holds (which is given by $SE' - SE$) and for each x_i making the condition false, which can be done by either deleting it from SE' (to make $x_i \in SE'$ false) or adding it to SE (to make $x_i \notin SE$ false). As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)}) =$
$\{c_t \leftarrow$!−(x, SE)$\}$

Proof: From the above plan type definition (and the plans for *!−(x,SE)*), any action sequence *as* in AS contains actions that remove x from SE.

Constraint c is violated if and only if x belongs to SE. Therefore, any action sequence that correctly (and minimally) repairs c must ensure that *SE→excludes(x)* which necessarily involves removing x from SE. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from

the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}excludesAll(SE')}) =$
$\{c_t \leftarrow$ for each x in SE $\cap$ SE' $!c'_t$(x),
c'_t(x) $\leftarrow !-(x, SE')$,
c'_t(x) $\leftarrow !-(x, SE)\}$

Proof: Let $\{x_1, \ldots, x_n\} = SE' \cap SE$ (where we know that $n > 0$, else the constraint is already true), and, because it is a set, we also know that $x_i \neq x_j$ for $i \neq j$.

From the above plan type definitions, any action sequence as in AS contains actions that for each x_i either remove it from SE' or remove it from SE.

The above constraint can be written as:

$$c \stackrel{\text{def}}{=} \forall\, x \in SE' \bullet x \notin SE$$

Assume that c is violated, i.e. $\neg$ c is true, expressed as follows.

$$\begin{aligned} \neg\, c &\stackrel{\text{def}}{=} \neg\, (\forall\, x \in SE' \bullet x \notin SE) \\ &\stackrel{\text{def}}{=} \exists\, x \in SE' \bullet x \in SE \end{aligned}$$

Now, $SE' \cap SE$ is the set of x_i for which the condition $x_i \in SE' \wedge x \in SE$ holds. Therefore, constraint c is violated (i.e. $\neg\, c$ is true) if and only if there is an x_i which is in both SE' and SE. Therefore to fix c (minimally) must ensure that no such x exists. We do this by finding all x_i for which the condition holds (which is given by $SE' \cap SE$) and for each x_i making the condition false, which can be done by either deleting it from SE' (to make $x_i \in SE'$ false) or deleting it from SE (to make $x_i \in SE$ false). As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs

c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty()}) =$
$\{c_t \leftarrow$ for each x in SE !−(x, SE)$\}$

Proof: Let $\{x_1, \ldots, x_n\} = SE$ (where we know that $n > 0$, else the constraint is already true), and, because it is a set, we also know that $x_i \neq x_j$ for $i \neq j$.

From the above plan type definitions, any action sequence as in AS contains actions that for each x_i remove it from SE.

Constraint c is violated (i.e. $\neg\ c$ is true) if and only if the SE is not empty. Hence, the only minimal way to prevent ¬ c from being true (or c from being false) is: for each x_i in SE we delete it from SE. This is the minimal way of fixing c since it does not involve removing any redundant elements. This matches exactly with an action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}notEmpty()}) =$
$\{c_t$: x ∈ Type(SE) ← !+(x, SE),
$c_t \leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$

Proof: The first plan type has the context condition $x \in Type(SE)$, which generates a number of plan instances corresponding to each entity in the set of $Type(SE)$. The second plan type does not have any context condition, thus it corresponds to exactly one plan instance. From the above plan type definitions, any action sequence as in AS contains actions that add an existing entity to SE (the first plan), or that create a new entity of the same type as entities contained in the set SE (the second plan).

Constraint c is violated if and only if the set SE is empty. Hence, the only minimal way to make c true is adding any entity that has the same type as entities contained in the set SE. This is the minimal way of fixing c since it does not involve adding any redundant elements. This can be done by either adding an existing entity to SE or creating a new entity of the same type as SE and adding it to SE. This matches exactly with an action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates

a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}forAll(c1)}) =$
$\{c_t \leftarrow$ for each x in SE if $\neg$ c1(x) then $!c'_t$(x),
c'_t(x) $\leftarrow$!$-$(x, SE),
c'_t(x) $\leftarrow$ $!c1_t$(x)$\}$

Proof: Let $\{x_1, \ldots, x_n\} = SE1$ be the set of entities in SE for which constraint $c1$ does not hold (where we know that $n > 0$, else the constraint is already true), and, because it is a set, we also know that $x_i \neq x_j$ for $i \neq j$.

From the above plan type definitions, any action sequence as in AS contains actions that for each x_i either remove it from SE or make $c1(x_i)$ true.

The constraint c can be written as:

$$c \stackrel{\text{def}}{=} \forall\, x \in SE \bullet c1(x)$$

Assume that c is violated, i.e. $\neg$ c is true, expressed as follows.

$$\begin{aligned} \neg\, c &\stackrel{\text{def}}{=} \neg\, (\forall\, x \in SE \bullet c1(x)) \\ &\stackrel{\text{def}}{=} \exists\, x \in SE \bullet \neg\, c1(x) \end{aligned}$$

Constraint c is violated if and only if SE contains an entity that constraint $c1$ does not hold. Hence, the only minimal way to prevent c from being false (i.e. make c true) is for each such x_i in SE we either delete it from SE (to make $x_i \in SE$ false) or make $c1(x_i)$ true (to make $\neg\, c1(x_i)$ false). This is the minimal way of fixing c since it does not involve removing any redundant elements. This matches exactly with an action sequence as in AS. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated

by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}exists(c1)}) =$
$\{c_t$: x ∈ SE ← !$c1_t$(x),
c_t : x ∈ Type(SE) ∧ x ∉ SE ← !+(x, SE) ; !$c1_t$(x),
c_t ← Create x : Type(SE) ; !+(x, SE) ; !$c1_t$(x),
$c1_t$(x) : c1(x) ∨ c ← true $\}$

Proof: It is noted that the fourth plan makes $c1(x)$ true if needed, otherwise (i.e. $c1(x)$ or c already holds) it does nothing.

Let $\{x_1, \ldots, x_n\} = \mathit{Type}(SE)$ be the set of existing entities that have the same type as entities in the set SE, and $\{x_1, \ldots, x_k\} = SE$ be the set of entities in the set SE (where $k \leq n$, and noting that $SE \subseteq \mathit{Type}(SE)$).

From the above plan type definitions, any action sequence as in AS contains actions that either make $c1(x_i)$ true for any x_i (where $1 \leq i \leq k$), or add x_j to SE and make $c1(x_j)$ true if it does not hold (where $k < j \leq n$), or create a new entity x, add it to SE and make $c1(x)$ true.

Constraint c is violated if and only if SE does not contain an entity for which constraint $c1$ holds. To fix c (minimally) involves making the condition hold for some x, i.e. $x \in SE \wedge c1(x)$. There are three cases: (1) x is already in SE: make $c1(x)$ true; (2) $c1(x)$ is already true: add x to SE; (3) neither: make $c1(x)$ true and add x to SE (note that x can be existing entity or a new one). As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow these patterns, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow one of these patterns, each of which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

A.1.4 Constraints on non-Boolean-valued set expressions

In this section, we present proofs for basic constraints concerning set expressions that return non-Boolean values. The proofs generally follow an induction approach in which we assume

that the set of generated repair plans for adding (or removing) an entity to (or from) a derived set is correct and complete[9].

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() = m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$|m - s|$} $!c'_t$,
c'_t : m < s $\wedge$ x $\in$ SE $\leftarrow$!−(x, SE),
c'_t : m > s $\wedge$ x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE),
c'_t : m > s $\leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$
where s = SE→size()

Proof: We now examine the following cases.

Case 1: ($m < s$, i.e. SE is too large)

The second and third plan types for handling c'_t are not applicable since their context conditions do not hold. Only the first plan type for c'_t is applicable and generates a number of plan instances, each of which corresponds to an entity in *SE*. From the plan type definitions, any action sequence *as* in *AS* contains actions that remove $s - m$ entities from *SE*.

To fix c (minimally) must reduce the size of *SE* by removing $s - m$ elements. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case.

Case 2: ($m > s$, i.e. SE is too small)

In this case, the second and third plan types for handling c'_t are applicable but the first plan type is not. From the plan type definitions, any action sequence *as* in *AS* contains actions that add $m - s$ entities to *SE* by using an existing entity (not belonging to *SE*) or a newly created one.

To fix c (minimally) must increase the size of *SE* by adding $m - s$ elements which can be either existing or newly created. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we

[9] We prove this assumption in section A.3

have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() \geq m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$(m-s)$}$\}$ $!c'_t$,
c'_t : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!+(x, SE),
$c'_t \leftarrow$ Create x : Type(SE) ; !+(x, SE)$\}$
where s = SE$\rightarrow$size()

Proof: We know that $s < m$, else the constraint is true. From the plan type definitions, any action sequence *as* in *AS* contains actions that add $m - s$ entities to *SE* by using an existing entity (not belonging to *SE*) or a newly created one.

To fix c (minimally) must increase the size of *SE* by adding $m - s$ elements which can be either existing or newly created. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{SE{\rightarrow}size() \leq m}) =$
$\{c_t \leftarrow$ for each i in {1,2,...,$(s-m)$}$\}$ $!c'_t$,
c'_t : x $\in$ SE $\leftarrow$!$-$(x, SE)$\}$
where s = SE$\rightarrow$size()

Proof: We know that $s > m$, else the constraint is true. From the plan type definitions, any action sequence *as* in *AS* contains actions that remove $s - m$ entities from *SE*.

To fix c (minimally) must reduce the size of *SE* by removing $s - m$ elements. As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow this pattern, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally repair c in this case. Furthermore, we have argued that any action sequence that correctly and minimally repairs c must follow this pattern, which is generated

by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for repairing c in this case.

A.1.5 Boolean connectives

An OCL constraint is ultimately a combination (*and*, *or*, *not*, *xor* and *implies*) of basic constraints. We have proved that theorem 1 holds for all basic constraints. We now use that to prove, by induction, that theorem 1 holds for the basic connectives: *and*, *or*, and *not*. Since the other connections (*xor* and *implies*) can be written using the three basic connectives (i.e. *and*, *or*, and *not*), their proof can also be derived from the basic ones.

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ or\ c2}) =$
$\{c_t \leftarrow !c1_t,$
$c_t \leftarrow !c2_t\}$

Proof: Assume that theorem 1 holds for $\mathcal{R}(c1_t)$ and $\mathcal{R}(c2_t)$, i.e. both of them are correct and complete sets. Now we need to prove that it also holds for $\mathcal{R}(c_t)$.

As above, we have:

$$\mathcal{P}_t(c) \quad = \quad \{c_t : \neg\ c1 \leftarrow !c1_t, c_t : \neg\ c2 \leftarrow !c2_t\}$$

and we also have:

$$\mathcal{R}(c_t) = \mathcal{P}_t(c) \cup \mathcal{R}(c1_t) \cup \mathcal{R}(c2_t) \cup \ldots$$

Because of our induction assumption $\mathcal{R}(c1_t)$ is correct and complete, i.e. it contains plans that correctly (and minimally) fix $c1$, and similarly for $\mathcal{R}(c2_t)$ and $c2$. Therefore, plan $c_t : \neg$ c1 $\leftarrow !c1_t$ is able to repair c1 and plan $c_t : \neg$ c2 $\leftarrow !c2_t$ is able to repair c2. Since the constraint c holds if either of $c1$ or $c2$ holds, any plan that is able to fix $c1$ or $c2$ can fix c. As a result, we can conclude that $\mathcal{R}(c_t)$ contains plans that correctly fix c. These plans are also minimal because they do not contain redundant repair actions. For instance, plan $c_t : \neg$ c1 $\leftarrow !c1_t$ fixes only $c1$ when $c1$ is false, which just sufficiently repairs c without the need to fix $c2$.

We have proved that $\mathcal{R}(c_t)$ contains correct and minimum repair plans for c. Now we prove the completeness of the set $\mathcal{R}(c_t)$. Assume that there is a minimum plan P that fixes c and does not belong to $\mathcal{R}(c_t)$. Plan P must fix either $c1$ or $c2$ (but not both, otherwise it is not minimal), and without loss of generality we assume that P aims to fix $c1$. Therefore, plan P is also the minimum plan for fixing $c1$, which results in, due to the induction assumption, that P belongs to $\mathcal{R}(c1_t)$. Since $\mathcal{R}(c_t)$ contains $\mathcal{R}(c1_t)$, P also belongs to $\mathcal{R}(c_t)$, which contradicts our previous assumption. Hence, there does not exist any minimum plan P that fixes c and does not belong to $\mathcal{R}(c_t)$, i.e. the set $\mathcal{R}(c_t)$ is complete. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{c1\ and\ c2}) =$
$\{c_t : \neg\, \mathrm{c1} \wedge \mathrm{c2} \leftarrow !c1_t,$
$c_t : \neg\, \mathrm{c2} \wedge \mathrm{c1} \leftarrow !c2_t,$
$c_t : \neg\, \mathrm{c1} \wedge \neg\, \mathrm{c2} \leftarrow !c1_t$[10]$\}$

Proof: Firstly, the complete set of repair plans for making constraint c true is:

$$\begin{aligned}\mathcal{R}_t(c) \;=\; & \{\mathit{fixC}_t : c \leftarrow \mathit{true}, \mathit{fixC}_t : \neg\, c \leftarrow !c_t;\ !\mathit{fixC}_t\} \cup \mathcal{R}(c1_t) \cup \mathcal{R}(c2_t) \cup \\ & \{c_t : \neg\, c1 \wedge c2 \leftarrow !c1_t, c_t : \neg\, c2 \wedge c1 \leftarrow !c2_t, c_t : \neg\, c1 \wedge \neg\, c2 \leftarrow !c1_t\}\end{aligned}$$

Because of our induction assumption $\mathcal{R}(c1_t)$ is correct and complete, i.e. it contains plans that correctly (and minimally) fix $c1$, and similarly for $\mathcal{R}(c2_t)$ and $c2$.

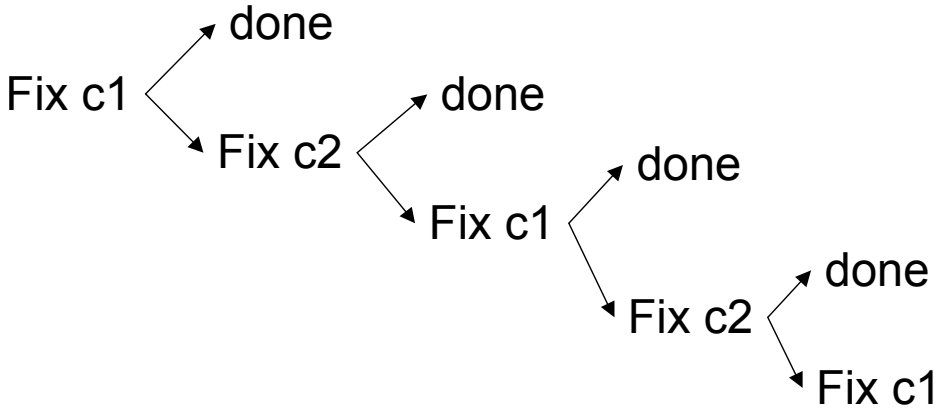


Figure A.1: An example of fixing constraint $c \stackrel{\text{def}}{=}$ c1 and c2

[10] In the case where both constraints $c1$ and $c2$ are false, without the loss of generality we only fix $c1$ first.

In order to fix constraint c, we begin by fixing either $c1$ or $c2$ (if initially both are false we can pick one of them, otherwise we begin with the one that is false). Then, depending on the precise fix used, either we are done (i.e. c is repaired) or the other constraint needs to be fixed. This situation repeats (potentially endlessly) and can be visualised as a tree (see figure A.1). The plans generated (including the $fixC_t$ plans) generate this sort of behaviour: repeatedly select a violated sub-constraint and fix it, repeating until "c1 and c2" (i.e. both constraints) are fixed. From this description it is easy to see that the process is correct: we continue until done, and at each point we fix something that needs fixing. Although it is not guaranteed that this process actually terminates[11], any finite action sequence generated by $\mathcal{R}$ is correct. With regard to minimality, it is noted that at each point we fix c1 or c2. The fix is only done if needed (otherwise we are done), and by induction the details of the fix are minimal.

We have proved that $\mathcal{R}(c_t)$ contains correct and minimum repair plans for c. Now we prove the completeness of the set $\mathcal{R}(c_t)$. Let as be an action sequence that is both correct (for fixing c) and minimal. Due to theorem 2 (on page 176), we can reorder as into *"as1 ; as2"* (i.e. action sequence $as1$ followed by action sequence $as2$) where $as1$ (correctly and minimally) repairs $c1$ and $as2$ (correctly and minimally) fixes $c2$. As argued earlier, *"as1 ; as2"* can be generated by $\mathcal{R}$. Hence, there does not exist any minimum action sequence as that fixes c and does not have a representative permutation that belong to $\mathcal{R}(c_t)$, i.e. the set $\mathcal{R}(c_t)$ is complete. ■

• $\mathcal{P}_t(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{not\ c1}) = \{\ c_t \leftarrow !c1_f\}$

Proof: Firstly, the complete set of repair plans for making constraint c true is:

$$\mathcal{R}(c_t) = \mathcal{P}_t(c) \cup \mathcal{R}(c1_f) \cup \ldots$$

Because of our induction assumption $\mathcal{R}(c1_f)$ is correct and complete, i.e. it contains plans that correctly (and minimally) make $c1$ false. Therefore, plan $c_t \leftarrow !c1_f$ is able to make c1 false. Since the constraint c holds if $c1$ is false, any plan that is able to make $c1$ false can fix c. As a result, we can conclude that $\mathcal{R}(c_t)$ contains plans that correctly (and

[11] If "c1 and c2" is satisfiable then it is possible for this process to terminate. In addition, our plan selection mechanism (discussed in chapter 7) is able to find terminating action sequences.

minimally) fix c.

We have proved that $\mathcal{R}(c_t)$ contains correct and minimum repair plans for c. Now we prove the completeness of the set $\mathcal{R}(c_t)$. Assume that there is a minimum plan P that fixes c and does not belong to $\mathcal{R}(c_t)$. Plan P should aim to make $c1$ false. Therefore, plan P is also the minimum plan for making $c1$ false, which results in, due to the induction assumption, that P belongs to $\mathcal{R}(c1_f)$. Since $\mathcal{R}(c_t)$ contains $\mathcal{R}(c1_f)$, P also belongs to $\mathcal{R}(c_t)$, which contradicts our previous assumption. Hence, there does not exist any minimum plan P that fixes c and does not belong to $\mathcal{R}(c_t)$, i.e. the set $\mathcal{R}(c_t)$ is complete. ■

A.2 Proofs for generated repair plans for making a constraint false, i.e. $\mathcal{R}(c_f)$

Since many of the proof cases here are analogous to those for making constraint true, similar proof can be used to prove the theorem's correctness. We now examine only the cases that are not analogous. Note that we consider that the constraints currently hold and the generated plans are for making them false.

- $\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend = x}) =$
  $\{c_f \leftarrow$ Disconnect E and x (w.r.t *aend*)$\}$

Proof: From the rule definition, there is only one generated repair plan type which has no context condition, i.e. it is always applicable and generates a single plan instance. Therefore, AS has a single *as* which contains only one action that disconnects E and x (w.r.t. *aend*).

Constraint c holds if and only if E is connected to x with respect to *aend*. The only minimal way of making c false is disconnecting E and x (w.r.t. *aend*). As can be seen from the plan generated by $\mathcal{R}$ in this case, the generated plan exactly follows this pattern, and thus the action sequence resulting from the plan generated by $\mathcal{R}(c_t)$ correctly and minimally make c false in this case. Furthermore, we have argued that any action sequence that correctly and minimally make c false must disconnect E and x (w.r.t. *aend*), and this is exactly what is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for making c false in this case. ■

- $\mathcal{P}_f(\mathbf{c} \stackrel{\text{def}}{=} \mathbf{E.aend1.aend2 = x}) =$
  $\{c_f \leftarrow$ Disconnect E.aend1 and x (w.r.t *aend*2),
  $c_f \leftarrow$ Disconnect E and E.aend1 (w.r.t. *aend*1)$\}$

Proof: This rule assumes that $E.aend1$ leads to a single entity and so does $E.aend1.aend2$. From the rule definition, there are two generated repair plan types which have no context condition, i.e. they are always applicable and generate two plan instances. From the definitions of these plans, any action sequence *as* in AS contains actions that either disconnect $E.aend1$ and x (w.r.t. $aend2$) or disconnect E and $E.aend1$ (w.r.t. $aend1$).

Constraint c holds if and only if E is connected to x with respect to $aend1.aend2$, i.e. $E.aend1.aend2 = x$. That means E is currently connected to only a y (w.r.t. $aend1$) and y is currently connected to only x (w.r.t. $aend2$). It is emphasized that with respect to $aend1$, entity E is not connected to any other entity except y. Therefore, any action sequence that correctly (and minimally) makes c false must ensure that $E.aend1.aend2 \neq x$ which necessarily involves ensuring either that $E.aend1 \neq y$ *or* that $y.aend2 \neq x$. The former involves disconnecting E and y (w.r.t. $aend1$) and the latter involves disconnected y and x (w.r.t. $aend2$). As can be seen from the plans generated by $\mathcal{R}$ in this case, the generated plans exactly follow these possibilities, and thus the action sequences resulting from the plans generated by $\mathcal{R}(c_t)$ correctly and minimally make c false in this case. Furthermore, we have argued that any action sequence that correctly and minimally make c false must follow these two patterns, both of which is generated by $\mathcal{R}$. Hence, $\mathcal{R}$ generates a representative permutation for each correct and minimal action sequence for making c false in this case. ■

A.3 Rules for addition involving derived sets, i.e. $\mathcal{Q}^+$

In this section we prove that the set of repair plans for addition involving a derived set (i.e. $\mathcal{Q}^+$), which are produced by our the repair plan generator based on the translation schemas described in figure 6.14 (on page 148), is correct and complete. That is, it generates all correct and minimal action sequences, and does not generate any incorrect action sequences, and all generated action sequences are minimal.

Similar to the previous proofs, we examine the repair plan types derived in each rule. We need to show that $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to a (derived) set. First, we examine a set of possible action sequences AS obtained from instantiating and resolving the plan types generated for a given set expression. We then identify all possible minimal ways of adding an entity to that set expression. Finally, we argue that the set of action sequences resulting from the generated repair plans for each constrain are precisely the possible minimal action sequences

for adding an entity to the set.

Furthermore, we will follow an induction approach. Firstly, we correctness for for the base cases SE $\stackrel{\text{def}}{=}$ E.aend and SE $\stackrel{\text{def}}{=}$ E.aend1.aend2. For sets that are derived from another set, for example SE $\stackrel{\text{def}}{=}$ S1→union(S2), we assume that generated action sequences for addition and deletion involving $S1$ and $S2$ are correct. We then prove, by induction, correctness for addition (and deletion) involving SE.

• $\mathcal{Q}^+$(**SE** $\stackrel{\text{def}}{=}$ **E.aend**[12]) =
{+(x, SE) ← Connect E and x (w.r.t *aend*)}

Proof: Since the above plan type has no context condition, it corresponds to exactly one plan instance. Therefore, AS contains a single action sequence *as* that has one action which connects E to x with regard to association end *aend*.

Note that $E.aend$ leads the set of entities that E are connected to with regard to *aend*. Hence, there is only one minimal way of adding an entity x to this set: connecting x with E (w.r.t. *aend*). This matches exactly with the action sequence *as* in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

• $\mathcal{Q}^+$(**SE** $\stackrel{\text{def}}{=}$ **E.aend1.aend2**) =
{+(x, SE) : isSingle(E.aend1) ∧ E.aend1 = y ← !+(x, y.aend2),
+(x, SE) : isSingle(E.aend1) ∧ x ∈ y.aend2 ← !($E.aend1 = y)_t$,
+(x, SE) : isSingle(E.aend1) ∧ y ∈ Type(E.aend1) ∧ y ≠ E.aend1 ∧ x ∉ y.aend2
← !($E.aend1 = y)_t$; !+(x, y.aend2),
+(x, SE) : isSingle(E.aend1) ← Create y : Type(E.aend1) ; !($E.aend1 = y)_t$; !+(x, y.aend2),
+(x, SE) : ¬ isSingle(E.aend1) ∧ y ∈ E.aend1 ← !+(x, y.aend2),
+(x, SE) : ¬ isSingle(E.aend1) ∧ x ∈ y.aend2 ← !+(y, E.aend1),
+(x, SE) : ¬ isSingle(E.aend1) ∧ y ∈ Type(E.aend1) ∧ y ∉ E.aend1 ∧ x ∉ y.aend2
← !+(y, E.aend1) ; !+(x, y.aend2),
+(x, SE) : ¬ isSingle(E.aend1)
← Create y : Type(E.aend1) ; !+(y, E.aend1) ; !+(x, y.aend2)}

Proof: We consider the following cases:

[12] $E.aend$ here returns a set of entities that are connected to E with regard to association end *aend*.

Case 1: $E.aend1$ leads to a single entity[13]

In this case, the first four plan types are applicable, which generate a number of plan instances. According to the plan definitions, any action sequence as in AS contains actions that either add x to the set $y.aend2$, i.e. $+(x, y.aend2)$, where y is an existing entity for which $E.aend1 = y$ holds (as in the first plan), or make $E.aend1 = y$ true for some y for which $x \in y.aend2$ holds (as in the second plan), or make $E.aend1 = y$ true and add x to $y.aend2$ where y is an existing entity or a newly created one. Making $E.aend1 = y$ true (or adding x to $y.aend2$) can yield further action sequences as previously discussed for making a constraint in the form of $E.aend = x$ true (or addition involving a derived set in the form of $E.aend$) . We have also proved that such action sequences are complete, correct and minimal. We use this result to prove by induction that repair plans posting the events of making $E.aend1 = y$ true (or adding x to $y.aend2$) also yield complete, correct, and minimal action sequences.

On the other hand, $E.aend1.aend2$ is the set of entities that $E.aend1$ is connected to with respect to $aend2$. Hence, the only minimal way of adding x into this set is: (i) adding x to $y.aend2$ and (ii) making $E.aend1 = y$ true for some y. Depending on the choice of y we may need to do either (i) – for y that E is currently connected to w.r.t. $aend1$, or (ii) – for y that is currently connected to x w.r.t. $aend2$, or both of them – for another existing entity y or a newly created one. Hence, the minimal way of adding x to the set $E.aend1.aend2$ matches exactly with an action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case.

Case 2: $E.aend1$ leads to multiple entities

In this case, $E.aend1$ is a set of entities. The proof is similar to the first case.

• $\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S{\rightarrow}select(c)})$[14] =
{ $+(x, SE)$: x ∈ S ← !c_t(x),
$+(x, SE)$: x ∉ S ∧ c(x) ← !+(x, S),

[13]Note that $E.aend1$ is not null, otherwise the set $E.aend1.aend2$ is not valid since "any property call applied on null results in OclInvalid, except for the operation oclIsUndefined()" [Object Management Group, 2006].

[14]Since there is not much difference in plans for SE $\stackrel{\text{def}}{=}$ S→reject(c), we can use a similar proof for them.

$+(x, SE)$: x $\notin$ S $\wedge$ $\neg$ c(x) $\leftarrow$!+(x, S) ; !c_t(x)$\}$

There are several cases that we need to consider.

Case 1: Entity x belongs to the set S

In this case, only the first plan type is applicable, which corresponds to a single plan instance. Any action sequence *as* in AS contains actions that make $c(x)$ true. Which actions are contained in such action sequences depends on the nature of constraint c. We, however, have proved previously that for any given constraint c (that we support), the generated action sequences for making c true are correct, complete, and minimal. As a result, by induction we can conclude that AS is also correct, complete, and minimal.

We know that SE is the set of entities in S for which the constraint c holds. Entity x belongs to S but is not contained in SE if and only if $c(x)$ does not hold. Therefore, the only minimal way to add x to SE is to make $c(x)$ true. This is the minimal way of adding x to SE since it does not involve adding any redundant elements. This matches exactly with an action sequence *as* in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case.

Case 2: Entity x does not belong to the set S

In this case, we consider two other cases: $c(x)$ holds and $c(x)$ does not hold. In the case where $c(x)$ holds the only minimal way of adding x to SE is adding x to S. Meanwhile, in the case where $c(x)$ does not hold, the only minimal way of adding x to SE is adding x to S and making $c(x)$ true. For either of these cases, we use a similar proof as in the first case. ■

• $\mathcal{Q}^+$(**SE** $\stackrel{\text{def}}{=}$ **S→excluding(e)**)[15] =

$\{+(x, SE)$: x $\notin$ S $\wedge$ x $\neq$ e $\leftarrow$!+(x, S)$\}$

Proof: From the plan definitions, in the case in which x belongs to S or x is e, then there is no applicable plan and consequently no action sequence for adding x to SE. In contrast, where x does not belong to S and x is different from e, the plan type is applicable and corresponds exactly to one plan instance. Hence, any action sequence *as* in AS contains actions that add x to S.

[15] A similar proof can be used for SE $\stackrel{\text{def}}{=}$ S→including(e)

From its definition, SE is the set of entities in S minus the entity e. If x is e, then we cannot add x to SE because SE always does not contain x. If x is not equal to e but x belongs to S, then x also belongs to SE and consequently adding x to SE results in no effect. However, from our perspective this case does not occur since we assume that when we post the event to add an entity to a set, the set must currently not contain the entity. Now where x does not belong to S and x is different from e, then the only minimal way to add x to SE is adding x to S. This is the minimal way since it does not involve adding any redundant entities. This matches exactly with an action sequence in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

• $\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1 \rightarrow union(S2)}) =$
{ $+(x, SE)$: x $\notin$ S1 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\notin$ S2 $\leftarrow$!+(x, S2)}

Proof: It is noted that x is neither in $S1$ nor in $S2$, otherwise x is already in SE. Hence, from the plan definitions both plan types are applicable. As a result, any action sequence as in AS contains actions that either add x to $S1$ or add x to $S2$.

From its definition, SE is the union of entities in $S1$ and $S2$, i.e. any entity that is in SE must be in either $S1$ or $S2$ or both of them. Therefore, the only minimal way to add x to SE is adding x to either $S1$ or $S2$. This is the minimal way since it does not involve adding any redundant entities or adding x to both $S1$ and $S2$. This matches exactly with an action sequence in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

• $\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1 \rightarrow intersection(S2)}) =$
{ $+(x, SE)$: x $\notin$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S2),
$+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S1) ; !+(x, S2)}

Proof: It is noted that x is not in both $S1$ and $S2$, otherwise x is already in SE. This leaves us three cases to be considered: x is in $S1$ but not in $S2$, x is in $S2$ but not in $S1$,

and x is in neither $S1$ nor $S2$. For the first two cases, without the loss of generality we can consider only one of the first two cases. For the third case, we can use a similar proof. Hence, we consider here only the first case, i.e. x is in $S1$ but not in $S2$.

In this case, only the second plan type is applicable. From the plan's definition, any action sequence *as* in AS contains actions that add x to $S2$.

From its definition, SE is the intersection of entities in $S1$ and $S2$, i.e. any entity that is in SE must be in both $S1$ and $S2$. As x is already in $S1$, the only minimal way to add x to SE is adding x to $S2$. This is the minimal way since it does not involve adding any redundant entities. This matches exactly with an action sequence in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

• $\mathcal{Q}^+(\mathbf{SE} \stackrel{\text{def}}{=} \mathbf{S1 - S2}) =$
$\{ +(x, SE) : \text{x} \notin \text{S1} \wedge \text{x} \in \text{S2} \leftarrow !+(\text{x, S1}) \; ; !-(\text{x, S2}),$
$+(x, SE) : \text{x} \notin \text{S1} \wedge \text{x} \notin \text{S2} \leftarrow !+(\text{x, S1}),$
$+(x, SE) : \text{x} \in \text{S1} \wedge \text{x} \in \text{S2} \leftarrow !-(\text{x, S2})\}$

Proof: It is noted that x is not in SE, i.e. x must either not be in $S1$, or, if it is in $S1$, it must also be in $S2$. This leaves us three cases to be considered: x is not in $S1$ but in $S2$, x is in both $S1$ and $S2$, and x is in neither $S1$ nor $S2$. We consider here only the first case, i.e. x is not in $S1$ but in $S2$. For the remaining two cases, a similar proof can be applied.

In this case (i.e. x is not in $S1$ but in $S2$), only the first plan type is applicable. From the plan's definition, any action sequence *as* in AS contains actions that add x to $S1$ and remove x from $S2$.

From its definition, SE contains the substraction of entities in $S1$ and $S2$, i.e. any entity that is in SE must be in $S1$ but not $S2$. As x is not in $S1$ but already in $S2$, the only minimal way to add x to SE is adding x to $S1$ and removing x from $S2$. This is the minimal way since it does not involve adding or removing any redundant entities. This matches exactly with an action sequence in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^+$. Hence, $\mathcal{Q}^+$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

• $\mathcal{Q}^{+}$(**SE** $\stackrel{\text{def}}{=}$ **S1→symmetricDifference(S2)**) =
{ $+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S1),
$+(x, SE)$: x $\notin$ S1 $\wedge$ x $\notin$ S2 $\leftarrow$!+(x, S2),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!−(x, S1),
$+(x, SE)$: x $\in$ S1 $\wedge$ x $\in$ S2 $\leftarrow$!−(x, S2)}

Proof: From its definition, SE contains all the entities that are either in $SE1$ or $SE2$ but not in both, i.e. any entity in SE must be either in $SE1$ or $SE2$ but not in both of them.

This leaves us two cases to be considered: x is in both $S1$ and $S2$, and x is in neither $S1$ nor $S2$ (for the other cases, x is in SE and consequently adding x to SE will result in no effect). We consider here only the first case, i.e. x is in neither $S1$ nor $S2$. For the other case, a similar proof can be applied.

In the case for which x is not in either $S1$ or $S2$, only the first two plan types are applicable. From the plan's definition, any action sequence as in AS contains actions that either add x to $S1$ or add x to $S2$.

Based on the definition of SE and because x is not in either $S1$ and $S2$, the only minimal way to add x to SE is either adding x to $S1$ or adding x to $S2$. This is the minimal way since it does not involve adding or removing any redundant entities. This matches exactly with an action sequence in AS. Furthermore, we have argued that any action sequence that correctly and minimally add x to SE must follow this pattern, which is generated by $\mathcal{Q}^{+}$. Hence, $\mathcal{Q}^{+}$ generates a representative permutation for each correct and minimal action sequence for adding an entity to SE in this case. ■

A.4 Rules for deletion involving derived sets, $\mathcal{Q}^{-}$

Since many of the proof cases here are analogous to those for addition involving a derived set (i.e. $\mathcal{Q}^{+}$), similar proof can be used to prove the theorem's correctness. We now examine only the cases that are not analogous.

• $\mathcal{Q}^{-}$(**SE** $\stackrel{\text{def}}{=}$ **E.aend**) =
{$-(x, SE)$ $\leftarrow$ Disconnect E and x (w.r.t *aend*)}

Proof: Since the above plan type has no context condition, it corresponds to exactly one plan instance. Therefore, AS contains a single action sequence as that has one action which disconnects E from x with regard to association end $aend$.

Note that $E.aend$ leads to the set of entities that E is connected to with regard to $aend$. Hence, there is only one minimal way of removing an entity x from this set: disconnecting x from E (w.r.t. $aend$). This matches exactly with the action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally remove x from SE must follow this pattern, which is generated by $\mathcal{Q}^-$. Hence, $\mathcal{Q}^-$ generates a representative permutation for each correct and minimal action sequence for removing an entity from SE in this case. ■

• $\mathcal{Q}^-$(**SE** $\stackrel{\text{def}}{=}$ **E.aend1.aend2**) =
{$-(x, SE)$: isSingle(E.aend1) $\wedge$ E.aend1 = E1
$\leftarrow$ Disconnect E1 and x (w.r.t $aend2$),
$-(x, SE)$: isSingle(E.aend1) $\wedge$ E.aend1 = E1
$\leftarrow$ Disconnect E and E1 (w.r.t $aend1$),
$-(x, SE)$: $\neg$ isSingle(E.aend1) $\wedge$ E.aend1 = SE1
$\leftarrow$ for each y in SE1 if x $\in$ y.aend2 then !aux(x, SE, y),
$aux(x, SE, y) \leftarrow$ Disconnect y and x (w.r.t. $aend2$),
$aux(x, SE, y) \leftarrow$ Disconnect E and y (w.r.t. $aend1$)}

Case 1: $E.aend1$ leads to a single entity

In this case, the first two plan types are applicable, which generate a number of plan instances. According to the plan definitions, any action sequence as in AS contains actions that either disconnect $E1$ and x (w.r.t. $aend2$) or disconnect E and $E1$ (w.r.t. $aend1$), where $E1 = E.aend1$ (i.e. E is currently connected to $E1$ w.r.t. $aend1$).

On the other hand, $E.aend1.aend2$ is the set of entities that $E.aend1$ (i.e. $E1$) is connected to with respect to $aend2$. Hence, the only minimal way of removing x from this set is either disconnecting $E1$ and x (w.r.t. $aend2$), or disconnecting E and $E1$ (w.r.t. $aend1$) – so that $E.aend1$ will become null and $E.aend1.aend2$ is invalid. Therefore, the minimal way of removing x from the set $E.aend1.aend2$ matches exactly with an action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally remove x from SE must follow this pattern, which is generated by $\mathcal{Q}^-$. Hence, $\mathcal{Q}^-$ generates a representative permutation for each correct and minimal action sequence for removing an entity from SE in this case.

Case 2: $E.aend1$ leads to a set of entities

In this case, the third plan type is applicable. Let $\{y_1, \ldots, y_n\} = SE2$ be the set of entities that E is connected to with regard to $aend1$, and that are connected to x (w.r.t. $aend2$), i.e. $x \in y_i.aend2$ (see figure A.2). From the above plan type definitions, any action sequence as in AS contains actions that for each y_i either disconnect it and x (w.r.t. $aend2$) or disconnect E and it (w.r.t. $aend1$).

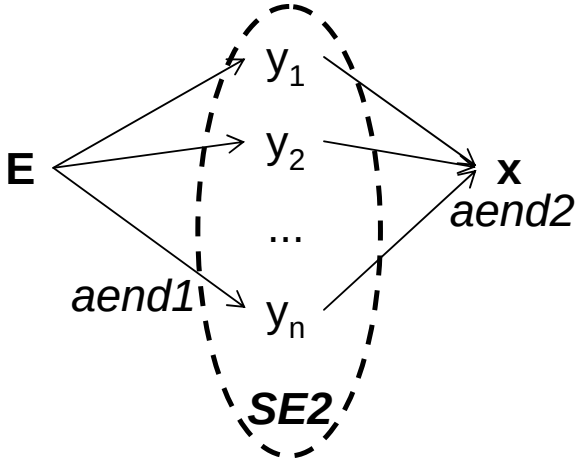


Figure A.2: An example showing how E is connected to x (w.r.t. $aend1.aend2$)

On the other hand, $E.aend1.aend2$ is the set of entities that E is connected to with respect to $aend1.aend2$, i.e. for x to be in $E.aend1.aend2$ there needs to be a path from E to x that follows $aend1$ and then $aend2$. Now, since $E.aend1$ can be a set, in fact there are multiple possible paths. Hence, to remove x from the set $E.aend1.aend2$ we need, for each such path, to break the path. $SE2$ is the set of middle points of these paths, from E to x (see figure A.2). For each middle point, y_i, we need to break the path, which can be done by either disconnecting E and y_i (w.r.t. $aend1$), or by disconnecting y_i and x (w.r.t. $aend2$). Therefore, the minimal way of removing x from the set $E.aend1.aend2$ matches exactly with an action sequence as in AS. Furthermore, we have argued that any action sequence that correctly and minimally remove x from SE must follow this pattern, which is generated by $\mathcal{Q}^-$. Hence, $\mathcal{Q}^-$ generates a representative permutation for each correct and minimal action sequence for removing an entity from SE in this case. ■

Bibliography

A. Abran and K. Nguyen. Measurement of the maintenance process from a demand-based perspective. *Software Maintenance and Evolution: Research and Practice*, 5(2):63–90, 1993.

E. Al-Hashel, B. M. Balachandran, and D. Sharma. A comparison of three agent-oriented software development methodologies: ROADMAP, Prometheus, and MaSE. In B. Apolloni, R. J. Howlett, and L. C. Jain, editors, *KES (3)*, volume 4694 of *Lecture Notes in Computer Science*, pages 909–916. Springer, 2007. ISBN 978-3-540-74828-1.

J. A. T. Álvarez, V. Requena, and J. L. Fernández. Emerging OCL tools. *Software and System Modeling*, 2(4):248–261, 2003.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521880386, 9780521880381.

T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062534.

D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1113034.1113040.

R. Arnold and S. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996. ISBN 978-0-8186-7384-9.

R. S. Arnold. An introduction to software restructuring. In R. S. Arnold, editor, *Tutorial on Software Restructuring*. IEEE Press, 1986.

R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8186-4600-4.

D. Astels. Refactoring with UML. In *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Process in Software Engineering*, pages 67–70, 2002. Alghero, Sardinia, Italy.

C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231149.

R. Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. ISBN 0-89791-391-4.

K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, 1999. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.796139.

S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 10–15, New York, NY, USA, 2006. ACM. ISBN 1-59593-303-4. doi: http://doi.acm.org/10.1145/1160633.1160938.

K. H. Bennett. Software maintenance: A tutorial. In R. H. Thayer and M. J. Christensen, editors, *Software Engineering, Volume 1: The Development Process*, volume 1. IEEE Computer Society Press, 3rd edition edition, November 2005.

K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 73–87. ACM Press, Limerick, Ireland, 2000.

F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems. The Agent-Oriented Software Engineering Handbook.* Kluwer Publishing, 2004. ISBN 1-4020-8057-3.

C. Bernon, M. Cossentino, and J. Pavón. Agent-oriented software engineering. *Knowledge Engineering Review*, 20(2):99–116, 2006. ISSN 0269-8889.

X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 511–520, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368158.

J. Blythe. Decision-theoretic planning. *AI Magazine*, 20(2):37–54, 1999.

J.-P. Bodeveix, T. Millan, C. Percebois, C. L. Camus, P. Bazex, and L. Feraud. Extending OCL for verifying UML models consistency. In Kuzniarz et al. [2002], pages 75–90.

M. Boger, T. Sturm, and P. Fragemann. Refactoring browser for UML. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 366–377, London, UK, 2003. Springer-Verlag. ISBN 3-540-00737-7.

S. A. Bohner. Software change impacts - an evolving perspective. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 263–272, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1819-2.

S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*, chapter An Introduction to Software Change Impact Analysis, pages 1–26. IEEE Computer Society Press, 1996.

R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

J. M. Bradshaw. An introduction to software agents. In J. M. Bradshaw, editor, *Software Agents*, pages 3–46. AAAI Press / The MIT Press, 1997.

M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987. ISBN 978-1575861920.

M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2304-8. doi: http://dx.doi.org/10.1109/CSMR.2005.1.

B. Breech, M. Tegtmeyer, and L. Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 55–65, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2354-4. doi: http://dx.doi.org/10.1109/ICSM.2006.33.

P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004. ISSN 1387-2532. doi: http://dx.doi.org/10.1023/B:AGNT.0000018806.20944.ef.

R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sowka. Automated impact analysis of UML models. *Journal of Systems and Software*, 79(3):339–352, March 2006. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2005.05.001.

J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(5): 309–332, September-October 2005.

B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. BDI-Agents for agile goal-oriented business processes. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 37–44, Estoril, Portugal, May 2008.

P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI agents in functional clusters. In *Agent Theories, Architectures, and Languages (ATAL-99)*, pages 277–289. Springer-Verlag, 2000. LNCS 1757.

J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321737.

J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *Proceedings of 2008 IEEE International Conference on Software Testing Verification and Validation, Workshop on Model Driven Engineering*, pages 73–80, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3388-9. doi: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2008.54.

F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modelling with a rule-based programming paradigm. *SIGMOD Rec.*, 19(2):225–236, 1990. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/93605.98732.

M. Cadoli, D. Calvanese, G. De Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by constraint programming. In *Proceedings of the CP 2004 Workshop on CSP Techniques with Immediate Application*, 2004.

S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transaction Database Systems*, 19(3):367–422, 1994. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/185827.185828.

N. Chapin. Do we know what preventive maintenance is? In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 15–17, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0.

N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001. ISSN 1040-550X.

M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming*, 45(2-3):155–174, 2002. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(02)00058-8.

E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.43044.

B. J. Clement and E. H. Durfee. Top-down search for coordinating the hierarchical plans of multiple agents. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 252–259. ACM Press, 1999. ISBN 1-58113-066-X. doi: http://doi.acm.org/10.1145/301136.301205.

M. Cossentino. From requirements to code with the PASSI methodology. In H.-S. B. and P. Giorgini, editors, *Agent-Oriented Methodologies*, pages 79–106. Idea Group Inc., 2005.

P. Cuesta, A. Gómez, and J. C. González. Agent oriented software engineering. In A. Moreno and J. Pavón, editors, *Issues in Multi-Agent Systems*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 1–31. Birkhauser Basel, December 2007.

K. H. Dam and M. Winikoff. Comparing agent-oriented methodologies. In Giorgini et al. [2004], pages 78–93.

S. Dance and M. Gorman. Intelligent Agents in the Australian Bureau of Meteorology. In *Challenges in Open Agent Systems Workshop at AAMAS02*, Bologna, Italy, July 2002.

A. Dasgupta and A. K. Ghose. CASO: a framework for dealing with objectives in a constraint-based extension to AgentSpeak(L). In *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, pages 121–126. Australian Computer Society, Inc., 2006. ISBN 1-920682-30-9.

S. A. DeLoach. Engineering organization-based multiagent systems. In A. F. Garcia, R. Choren, C. J. P. de Lucena, P. Giorgini, T. Holvoet, and A. B. Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV, Research Issues and Practical Applications*, volume 3914 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2005. ISBN 3-540-33580-3.

D. Dennett. *The Intentional Stance*. MIT Press, Cambridge, Mass., 1987. ISBN 978-0262540537.

L. Deruelle, M. Bouneffa, N. Melab, and H. Basson. A change propagation model and platform for multi-database applications. In *ICSM '01: Proceedings of the IEEE International*

Conference on Software Maintenance (ICSM'01), pages 42–51, Los Alamitos, CA, USA, 2001. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ICSM.2001.972710.

M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *ATAL '97: Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, pages 155–176, London, UK, 1998. Springer-Verlag. ISBN 3-540-64162-9.

K. Dohyung. Java MPEG player at http://peace.snu.ac.kr/dhkim/java/mpeg, 1999.

A. Egyed. Instant consistency checking for the uml. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 381–390, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134339.

A. Egyed. Fixing inconsistencies in UML models. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 292–301, Washington, DC, USA, May 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

A. Egyed and D. S. Wile. Support for managing design-time decisions. *IEEE Transactions on Software Engineering*, 32(5):299–314, 2006.

M. Elaasar and L. Briand. An overview of UML consistency management. Technical Report SCE-04-18, Carleton University, Department of Systems and Computer Engineering, 2004.

A. H. Elamy and B. H. Far. On the evaluation of agent-oriented software engineering methodologies: A statistical approach. In M. Kolp, B. Henderson-Sellers, H. Mouratidis, A. Garcia, A. Ghose, and P. Bresciani, editors, *Agent-Oriented Information Systems IV, 8th International Bi-Conference Workshop*, volume 4898 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2006. ISBN 978-3-540-77989-6.

G. Engels, J. M. Kuster, R. Heckel, and L. Groenewegen. Towards consistency-preserving model evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 129–132. ACM Press, 2002. ISBN 1-58113-545-9. doi: http://doi.acm.org/10.1145/512035.512066.

E. Ephrati, M. E. Pollack, and M. Milshtein. A cost-directed planner: Preliminary report. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 1223–1228, 1996.

K. Erol, J. Hendler, and D. S. Nau. HTN planning: complexity and expressivity. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, pages 1123–1128, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. ISBN 0-262-61102-3.

J. Ferber. *Multi-agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999. ISBN 978-0201360486.

S. Fickas, M. Feather, and J. Kramer, editors. *Proceedings of the Workshop on Living with Inconsistency*, Boston, USA, 1997.

A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992. URL http://dblp.uni-trier.de/db/journals/ijseke/ijseke2.html#FinkelsteinKNFG92.

A. Finkelstein, G. Spanoudakis, and D. Till. Managing interference. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 172–174, New York, NY, USA, 1996. ACM. ISBN 0-89791-867-3. doi: http://doi.acm.org/10.1145/243327.243646.

A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.310667.

R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In N. Z. G. Parikh, editor, *Tutorial on Software Maintenance*, pages 13–30. IEEE Computer Society Press, Los Alamitos, LA, 1982.

M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193, Berlin, Germany, 1996. Springer-Verlag.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995. ISBN 0201633612.

M. Georgeff. Service orchestration: The next big challenge. *DM Direct Special Report*, June 2006.

M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.

M. Gertz and U. W. Lipeck. An extensible framework for repairing constraint violations. In *Proceedings of the IFIP TC11 Working Group 11.5, First Working Conference on Integrity and Internal Control in Information Systems*, pages 89–111. Chapman & Hall, Ltd., 1997. ISBN 0-412-82600-3.

A. K. Ghose. Formal tools for managing inconsistency and change in RE. In *IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design*, pages 171–181, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0884-7.

P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors. *Agent-Oriented Information Systems, 5th International Bi-Conference Workshop, AOIS 2003, Melbourne, Australia, July 14, 2003 and Chicago, IL, USA, October 13th, 2003, Revised Selected Papers*, volume 3030 of *Lecture Notes in Computer Science*, 2004. Springer.

R. L. Glass. The naturalness of object orientation: Beating a dead horse? *IEEE Software*, 19(3):104, 2002. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2002.1003467.

J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11): 960–981, 1998. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.730545.

S. Gwizdala, Y. Jiang, and V. Rajlich. JTracker - a tool for change propagation in Java. In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 223–229. IEEE Computer Society, 26-28 March 2003. Benevento, Italy.

I. Hadar and U. Leron. How intuitive is object-oriented design? *Communications of the ACM*, 51(5):41–46, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1342327.1342336.

R. Haesen and M. Snoeck. Implementing consistency management techniques for conceptual modeling. In *Proceedings of the International Conference on the Unified Modeling Language 2004 (Workshop 7: Consistency Problems in UML-based Software Development)*, Lisbon, Portugal, October 10–15 2004.

J. Han. Supporting impact analysis and change propagation in software engineering environments. In *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, pages 172–182, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7840-2.

A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0.

D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS'06: Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2006. ISBN 3-540-45772-0.

B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group Publishing, 2005. ISBN 978-1591405818.

P. V. Hentenryck and V. Saraswat. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701–726, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242223.242279.

K. V. Hindriks, F. S. D. Boer, W. V. D. Hoek, and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999. ISSN 1387-2532. doi: http://dx.doi.org/10.1023/A:1010084620690.

J. F. Horty and M. E. Pollack. Evaluating new options in the context of existing plans. *Artificial Intelligence*, 127(2):199–220, 2001. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(01)00060-1.

M.-P. Huget and J. Odell. Representing agent interaction protocols with agent UML. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1244–1245, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 1-58113-864-4. doi: http://dx.doi.org/10.1109/AAMAS.2004.230.

IEEE. IEEE standard for software maintenance, IEEE Std 1219-1998. In *IEEE Standards Software Engineering, Volume Two: Process Standards*, volume 2. IEEE Press, New York, NY, 1999.

IEEE. International Standard - ISO/IEC 14764 IEEE Std 14764-2006. *Software Engineering, Software Life Cycle Processes, Maintenance (Revision of IEEE Std 1219-1998)*, 2006.

F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992. ISSN 0885-9000. doi: http://dx.doi.org/10.1109/64.180407.

ISO/IEC 14764. Information technology - software maintenance. ISO: Geneva, Switzerland, 1999.

I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 252–261. IEEE Computer Society, 2004. ISBN 0-7695-2213-0.

G. Jayatilleke, L. Padgham, and M. Winikoff. Component agent framework for non-experts (CAFnE) toolkit. In R. Unland, M. Calisti, and M. Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 169–195. Birkhauser Basel, 2005.

G. B. Jayatilleke. *A Model Driven Component Agent Framework for Domain Experts*. PhD thesis, RMIT University, Australia, March 2007.

N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367211.367250.

N. R. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. In F. J. Garijo and M. Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous*

Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99), volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999. URL http://citeseer.nj.nec.com/article/jennings00agentoriented.html.

N. R. Jennings and M. J. Wooldridge. Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany, 1998.

N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

M. Kajko-Mattsson. Preventive maintenance! do we know what it is? In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 12–14, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0.

B.-K. Kang and J. M. Bieman. A quantitative framework for software restructuring. *Journal of Software Maintenance*, 11(4):245–284, 1999. ISSN 1040-550X.

A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, MA, 2003. ISBN 978-0321194428.

J. Koskinen. Software maintenance costs, September 2004. http://users.jyu.fi/ koskinen/smcosts.htm.

A. Krishna, S. A. Vilkomir, and A. K. Ghose. Consistency preserving co-evolution of formal specifications and agent-oriented conceptual models. *Information and Software Technology*, 51(2):478–496, 2009. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2008.05.015.

P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.469759.

D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 202–211, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6330-8.

L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development*, 2002:06, Ronneby, 2002. Blekinge Institute of Technology.

L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2003, Modeling Languages and Applications. Workshop on Consistency Problems in UML-based Software Development II*, 2003:06, Ronneby, 2003. Blekinge Institute of Technology.

J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.

M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE, Special Issue on Software Evolution*, 68(9):1060–1076, September 1980.

M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X.

M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-442440-6.

M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8093-8.

B. P. Lientz and E. B. Swanson. *Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Publishing Company, Reading, MA, USA, 1980. ISBN 0201042053.

W. Liu, S. Easterbrook, and J. Mylopoulos. Rule based detection of inconsistency in UML models. In Kuzniarz et al. [2002], pages 106–123.

M. Ljungberg and A. Lucas. The OASIS air-traffic management system. In *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI '92)*, Seoul, Korea, 1992.

M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005. URL http://www.agentlink.org/roadmap/al3rm.pdf.

Luqi. A graph model for software evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.57627.

H. Malik and A. E. Hassan. Supporting software evolution using adaptive change propagation. In *ICSM '08: Proceedings of the 24th IEEE International Conference on Software Maintenance*, Beijing, China, September 2008.

I. Mathieson, S. Dance, L. Padgham, M. Gorman, and M. Winikoff. An open meteorological alerting system: Issues and solutions. In V. Estivill-Castro, editor, *Proceedings of the 27th Australasian Computer Science Conference*, pages 351–358, Dunedin, New Zealand, 2004.

E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 62–73, London, UK, 1999. Springer-Verlag. ISBN 3-540-66653-2.

E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 331–340. IEEE Computer Society, April 2006. doi: 10.1109/ASWEC.2006.26.

S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

T. Mens. Introduction and roadmap: History and challenges of software evolution. In T. Mens and S. Demeyer, editors, *Software Evolution*. Springer Berlin Heidelberg, 2008.

T. Mens and T. D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000. ISSN 0928-8910. doi: http://dx.doi.org/10.1023/A:1008765200695.

T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2004.1265817.

T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *Algebraic Description Techniques*, volume 4409, pages 111–127. Springer-Verlag, 2007.

T. Mens, R. V. D. Straeten, and J. Simmonds. A framework for managing consistency of evolving UML models. In H. Yang, editor, *Software Evolution with UML and XML*, pages 1–31. Idea Group Publishing, 2005. ISBN 1591404630.

T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199, pages 200–214. Springer-Verlag, October 2006.

D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.708567.

H. W. Miller. *Reengineering legacy software systems.* Digital Press, Newton, MA, 1998. ISBN 978-1555581954.

J. Miller and J. Mukerji. *MDA Guide Version 1.0.1.* Object Management Group, 2003.

G. Moerkotte and P. C. Lockemann. Reactive consistency control in deductive databases. *ACM Transaction Database Systems*, 16(4):670–702, 1991. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/115302.115298.

L. Monostori, J. Váncza, and S. Kumara. Agent based systems for manufacturing. *CIRP Annals-Manufacturing Technology*, 55(2):697–720, 2006.

M. G. Morris, C. Speier, and J. A. Hoffer. An examination of procedural and object-oriented systems analysis methods: Does prior experience help or hinder performance? *Decision Sciences*, 30(1):107–136, 1999.

H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the 22th International Conference on Software Engineering (Future of Software Engineering track)*, pages 47–60, New York, NY, USA, 2000. ACM. doi: http://doi.acm.org/10.1145/336512.336526.

S. Munroe, T. Miller, R. A. Belecheanu, M. Pěchouček, P. McBurney, and M. Luck. Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. *Knowledge Engineering Review*, 21(4):345–392, 2006. ISSN 0269-8889. doi: http://dx.doi.org/10.1017/S0269888906001020.

N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(98)00068-X.

D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/514183.514186.

C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 455–464. IEEE Computer Society, 2003. ISBN 0-7695-1877-X.

M. Nowostawski, M. Purvis, and S. Cranefield. A layered approach for modelling agent conversations. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, 5th International Conference on Autonomous Agents*, pages 163–170, Montreal, 2001.

B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.839317.

H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(2):205–244, 1995.

Object Management Group. Meta Object Facility Specification (MOF 1.4). http://www.omg.org/technology/documents/formal/mof.htm, 2002.

Object Management Group. Object Constraint Language (OCL) 2.0 Specification. http://www.omg.org/docs/ptc/03-10-14.pdf, 2006.

Object Management Group. UML 2.0 Superstructure and Infrastructure Specifications. http://www.omg.org/technology/uml/, 2004.

Object Management Group. Unified Modeling Language (UML) 1.4.2 specification (ISO/IEC 19501). http://www.omg.org/technology/uml/, 2005.

Object Management Group. XML Metadata Interchange (XMI) Specification (Version 2.0). http://www.omg.org/technology/documents/formal/xmi.htm, 2003.

J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, 2002. URL http://www.jot.fm/issues/issue_2002_05/column4.pdf.

J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *The Agent Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17. ICue Publishing, 2000.

W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

L. Padgham and M. Perepletchikov. Prioritisation mechanisms to support incremental development of agent systems. *International Journal of Agent-Oriented Software Engineering*, 1(3/4):477–497, 2007. ISSN 1746-1375. doi: http://dx.doi.org/10.1504/IJAOSE.2007.016269.

L. Padgham and M. Winikoff. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, Chichester, 2004. ISBN 0-470-86120-7.

L. Padgham, J. Thangarajah, and M. Winikoff. Tool support for agent development using the Prometheus methodology. In *First international workshop on Integration of Software Engineering and Agent Technology (ISEAT 2005)*, Melbourne, Australia, September 2005.

L. Padgham, M. Winikoff, S. DeLoach, and M. Cossentino. A unified graphical notation for AOSE. In M. Luck and J. J. Gomez-Sanz, editors, *Proceedings of the Ninth International Workshop on Agent Oriented Software Engineering*, pages 61–72, Estoril, Portugal, May 2008.

D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

M. Perepletchikov and L. Padgham. Use case and actor driven requirements engineering: An evaluation of modifications to prometheus. In M. Pechoucek, P. Petta, and L. Z. Varga, editors, *Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005, Budapest, Hungary, September 15-17, 2005, Proceedings*, volume 3690 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN 3-540-29046-X.

T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0471170011.

A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.

M. Polo, M. Piattini, and F. Ruiz, editors. *Advances in software maintenance management: Technologies and solutions*. Idea Group Publishing, U.S.A, 2003. ISBN 978-1591400479.

I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language, 6th International Conference*, volume 2863 of *Lecture Notes in Computer Science*, San Francisco , CA, USA, Oct 2003. Springer.

D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 960–967, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0. doi: http://doi.acm.org/10.1145/544862.544966.

V. Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, 2006.

V. Rajlich. Theory of data structures by relational and graph grammars. In *Proceedings of the Fourth Colloquium on Automata, Languages and Programming*, pages 391–411, London, UK, 1977. Springer-Verlag. ISBN 3-540-08342-1.

V. Rajlich. A methodology for incremental changes. In *Proceedings of the 2nd International*

Conference on eXtreme Programming and Flexible Process in Software Engineering, pages 10–13, Cagliary, Italy, May 2001.

V. Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 84–91. IEEE Computer Society, 1997. ISBN 0-8186-8013-X.

V. Rajlich. Software change and evolution. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 189–202, London, UK, 1999. Springer-Verlag. ISBN 3-540-66694-X. LNCS 2863.

V. Rajlich. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering*, 9(1-4):235–248, 2000. ISSN 1022-7091.

A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55. Springer-Verlag, 1996. ISBN 3-540-60852-4.

A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann, 1991. ISBN 1-55860-165-1.

A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

S. P. Reiss. Incremental maintenance of software artifacts. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi: http://dx.doi.org/10.1109/ICSM.2005.54.

G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.689399.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings Program Analysis for Software Technology (PASTE)*, pages 46–53, June 2001.

D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2): 25–31, 2006. doi: http://dx.doi.org/10.1109/MC.2006.58.

R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321118847.

S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231150.

W. Shen and D. H. Norrie. Agent-based systems for intelligent manufacturing: A state-of-the-art survey. *Knowledge and Information Systems*, 1(2):129–156, 1999.

Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.

G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In K. S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 24–29. World Scientific, 2001.

J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-983768-X.

R. V. D. Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1210–1217, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. doi: http://doi.acm.org/10.1145/1141277.1141564.

A. Sturm and O. Shehory. A framework for evaluating agent-oriented methodologies. In Giorgini et al. [2004], pages 94–109.

E. B. Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

A. A. Takang and P. A. Grubb. *Software maintenance : concepts and practice*. International Thomson Computer Press, London ; Boston, 2nd edition, 2003. ISBN 978-9812384263.

J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002*, pages 18–22. IOS Press, 2002.

Q.-N. N. Tran and G. C. Low. Comparison of ten agent-oriented methodologies. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, chapter Chapter XII, pages 341–367. Idea Group Publishing, 2005.

L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, March 2008. URL http://www.jot.fm/issues/issue_2008_03/article3/.

J. J. P. Tsai, T. Weigert, and H.-C. Jang. A hybrid knowledge representation as a basis of requirement specification and specification analysis. *IEEE Transactions on Software Engineering*, 18(12):1076–1100, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.184762.

S. Urban, A. Karadimce, and R. Nannapaneni. The implementation and evaluation of integrity maintenance rules in an object-oriented database. *Proceedings of the Eighth International Conference on Data Engineering*, pages 565–572, February 1992. doi: 10.1109/ICDE.1992.213152.

R. Van Der Straeten. *Inconsistency Management in Model-Driven Engineering*. PhD thesis, Vrije Universiteit Brussel, 2005.

R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logics to maintain consistency between UML models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, pages 326–340. Springer-Verlag, 2003. ISBN 3-540-20243-9. LNCS 2863.

A. van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In D. Tamzalit, editor, *Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE)*, pages 41–49. University of Nantes, 2007.

A. van Lamsweerde, E. Letier, and R. Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.730542.

H. Van Vliet. *Software engineering: principles and practice*. John Wiley & Sons, Inc., 2nd edition, 2001. ISBN 0471975087.

I. Vessey and S. A. Conger. Requirements specification: learning object, process, and data methodologies. *Communications of the ACM*, 37(5):102–113, 1994. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/175290.175305.

A. von Knethen and M. Grund. Quatrace: a tool environment for (semi-) automatic impact analysis based on traces. *ICSM'03: Proceedings of 19th International Conference on Software Maintenance*, pages 246–255, Sept. 2003. ISSN 1063-6773. doi: 10.1109/ICSM.2003.1235427.

R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. In *UML 2003: Modeling Languages and Applications. Workshop on Consistency Problems in UML-based Software Development II*, San Francisco, CA, USA, 2003.

J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321179366.

M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

M. Williamson. Optimal planning with a goal-directed utility model. In K. J. Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, pages 176–181. AAAI, 1994. URL citeseer.ist.psu.edu/williamson94optimal.html.

M. Winikoff. Defining syntax and providing tool support for agent uml using a textual notation. *International Journal of Agent-Oriented Software Engineering*, 1(2):123–144, 2007.

M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: the state of the art. In *Proceedings of the 1st international workshop on Agent-oriented software engineering (AOSE 2000)*, pages 1–28. Springer-Verlag New York, Inc., Limerick, Ireland, 2001.

M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

S. Yau, J. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 60–65, 1978.

S. S. Yau, R. A. Nicholl, J. J.-P. Tsai, and S.-S. Liu. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 14(8):1128–1144, 1988. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.7624.

F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, 2004. ISSN 1387-2532. doi: http://dx.doi.org/10.1023/B:AGNT.0000038028.66672.1e.

F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3): 317–370, 2003. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/958961.958963.

J. Zhang, Y. Lin, and J. Gray. *Model-Driven Software Development*, chapter Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine, pages 199–217. Springer Berlin Heidelberg, 2005.