Software Traceability for Multi-Agent Systems Implemented Using BDI Architecture

Gilberto Amado de Azevedo Cysneiros Filho

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at City University London

City University London
Department of Computing

June 2011

Volume 1

Contents

Volume 1

Volume 2

Figures

Tables

Acknowledgements

I would like to thank the examiners Peter Sawyer and Bill Karakostas for having so kindly accepted to take part of my viva voice examination and for the comments and suggestions. The quality of thesis would have suffered without their contribution.

Andre Zisman has been principal motivator actor of my work giving helpful feedback during the period of her supervision. It was also great value have written papers with her and with my co-supervisor George Spanoudakis.

Thank you to all colleagues from Department of Computing who I have had the good fortune to share a room with or work together with as visiting tutor. Especially, I would like to thank Michael Iossif, Mark Firman, Shant Narcessian, Waraporn Jirapathong, Khaled Mahub, Marcus Andrews, Olga Castilho, Thsiamo, Theoharris, Ricardo Contreras, and George Lekeas.

I would also like to thank the support and administrative team for all their help during all this period.

Thank you especially to my friends from London for the support and attention that made life easier and happier.

My greatest gratitude goes to my family that had suffered from my absence and for the support that they always gave in my life.

Finally, I would to thank you the "Lord" that without anything would not be possible.

Declaration

Abstract

The development of multi-agent software systems is considered a complex task due to (a) the large number and heterogeneity of documents generated during the development of these systems, (b) the lack of support for the whole development life-cycle by existing agent-oriented methodologies requiring the use of different methodologies, and (c) the possible incompleteness of the documents and models generated during the development of the systems.

In order to alleviate the above problems, in this thesis, a traceability framework is described to support the development of multi-agent systems. The framework supports automatic generation of traceability relations and identification of missing elements (i.e., completeness checking) in the models created during the development life-cycle of multi-agent systems using the Belief-Desire-Intention (BDI) architecture.

Traceability has been recognized as an important activity in the software development process. Traceability relations can guarantee and improve software quality and can help with several tasks such as the evolution of software systems, reuse of parts of the system, validation that a system meets its requirements, understanding of the rationale for certain design decisions, identification of common aspects of the system, and analysis of implications of changes in the system.

The traceability framework presented in this thesis concentrates on multi-agent software systems developed using *i** framework, Prometheus methodology, and JACK language. Here, a traceability reference model is presented for software artefacts generated when using *i** framework, Prometheus methodology, and JACK language. Different types of relations between the artefacts are identified. The framework is based on a rule-based approach to support automatic identification of traceability relations and missing elements between the generated artefacts. Software models represented in XML were used to support the heterogeneity of models and tools used during the software development life-cycle. In the framework, the rules are specified in an extension of XQuery to support (i) representation of the consequence part of the rules, i.e. the actions to be taken when the conditions are satisfied, and (ii) extra functions to cover some of the traceability relations being proposed and completeness checking of the models.

A prototype tool has been developed to illustrate and evaluate the work. The work has been evaluated in terms of recall and precision measurements in three different case studies. One small case study of an Automatic Teller Machine application, one medium case study of an Air Traffic Control Environment application, and one large case study of an Electronic Bookstore application.

Chapter 1 - Introduction

A multi-agent system consists of a system composed of several agents that are situated in an environment and that interact with each other and with their environment. Multi-agent systems have been proposed as a solution to implement complex systems that need to run in an environment that is open, distributed and highly interactive. An agent is defined by Wooldridge in (Wooldridge, et al., 1995), (Wooldridge, 2002) as a software component that is "situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives". Several types of software components fulfil this definition varying from daemons process in UNIX (Frisch, 2002) to complex decision making systems that control unmanned autonomous vehicles (Agent Oriented Software Limited, 2010).

An intelligent agent is an autonomous software component that is categorised to be pro-active, reactive, and social (Wooldridge, 2002). Pro-activeness means that the agent takes initiative in order to achieve its goals. Reactivity means that the agent perceives its environment and responds to its stimulus according to its goals. Social ability means that the agent will be able to communicate with other agents and have abilities such as co-operation, co-ordination, and negotiation.

Several architectures have been proposed to build multi-agent systems such as Jadex (Pokahr, et al., 2005), Jason (Bordini, et al., 2005), and JACK (Busetta, et al., 1999), (Howden, et al., 2001). Agent architectures can be classified in three categories: deliberative architectures, reactive architectures, hybrid architectures.

Reactive architectures do not maintain a symbolic representation of the environment and actions are performed using rules. Agents are situated in the environment and perceive the environment. Depending on the event that occurs in the environment a rule is executed and actions are performed.

In the deliberative architecture, a symbolic representation of the environment is created and the agent performs actions to manipulate these symbols. The actions performed are based on logical reasoning using theorem provers (Genesereth, et al., 1987). The drawback of this architecture is that it is difficult to represent the real world using a symbolic representation. Moreover, the use of logic reasoning to determine what action to perform is a very resource and time consuming task. Several multi-agent systems use a deliberative architecture to support reasoning and some of them are based on the BDI (Belief Desire Intention) architecture (Bratman, et al., 1988). Hybrid architectures combine deliberative and reactive behaviour. Examples of hybrid architectures are: TouringMachines, and INTERRRAP (Luck, et al., 2004).

BDI architectures have been proposed to address the problem of resource boundedness. The BDI architecture (Rao, et al., 1992) is one of the most successful architectures. The BDI architecture is founded on the philosophy theory of Bratman (Bratman, 1999) to explain human rationale action and it has been formalised by logic theory called LORA (Wooldridge, 2000) and BDI logic (Rao, et al., 1998). The BDI architecture has been implemented several times. Examples of implementation are: IRMA (Bratman, et al., 1988), PRS (Ingrand, et al., 1992), Jadex (Pokahr, et al., 2005), Jason (Bordini, et al., 2005) and JACK (Howden, et al., 2001), (Agent Oriented Software Limited, 2010).

Bratman et al. describe the Intelligent Resource-Bounded Machine Architecture (IRMA) that is the first implementation of BDI architecture (Luck, et al., 2004). The IRMA architecture addresses the problem of how an agent can select the best set of actions to carry out in order to achieve a goal when limited by resources such as the amount of time to take the decision.

The Procedural Reasoning System (PRS) is one of the most successful implementation of BDI architecture. The PRS architecture was used to build several applications such as a prototype system to manage the air traffic control of Sydney airport (Ljungberg, et al., 1992), (Rao, et al., 1995). The PRS system has been re-implemented and extended several times. The most known implementations are dMARS (d'Inverno, et al., 2004), JAM (Huber, 1999), JACK (Howden, et al., 2001), (Agent Oriented Software Limited, 2010), and Jadex (Pokahr, et al., 2005).

To support the development of multi-agent systems various methodologies have been proposed such as Prometheus (Padgham, et al., 2004), Tropos (Castro, et al., 2002), MaSE (DeLoach, 2001), and Gaia (Wooldridge, et al., 2000). These methodologies can be classified based on their origins. For instance, Tropos is based on requirements oriented methodologies and has its origins on *i** framework. Prometheus is based on object-oriented methodologies and its design phase is influenced by JACK. Luck et al. (Luck, et al., 2004) and Sudeikat et al. (Sudeikat, et al., 2004) classify agent oriented methodologies origins as object-oriented, knowledge engineering oriented, requirement engineering oriented, and of general category.

Despite advances in the area, the development of multi-agent systems is a complex task. As outlined in (Luck, et al., 2004), the difficulty to develop multi-agent systems are due to the (a) design of software systems that maintain a balance between proactive and reactive behaviour present in agents, (b) understanding of when agent approaches are appropriate, and (c) use of informal development techniques. In addition, (i) the large number and heterogeneity of documents generated during the development of multi-agent systems, (ii) the lack of support for the whole development life-cycle by existing agent-oriented methodologies requiring the use of different methodologies, and (iii) the possible incompleteness of the documents and models generated during the development of multi-agent systems contribute to the difficulties of developing such systems.

Moreover, the development of multi-agent systems produces a huge number of artefacts. Each artefact created can be related to several other artefacts. The relations between artefacts can be explicit or implicit. Explicit relations are concerned with the direct relation between two artefacts. For instance, artefact B depends on artefact A. Therefore, there is an explicit relation between the artefacts A and B. Implicit relations are concerned with indirect relations between two artefacts. For instance, artefact B depends on artefact A and artefact C depends on artefact B. Therefore there is an implicit relation between artefacts A and C.

Explicit relations are easier to maintain while implicit relations are difficult to maintain and to be found. Furthermore, multi-agents systems are normally developed by teams of analysts, developers, and programmers that are often distributed in different locations and use different tools, notations, and methodologies. The heterogeneity of people, tools, notations, and methodologies makes difficult to identify and understand the relations between the artefacts. In addition, it is not possible to guarantee completeness of the generated artefacts.

The need to understand the relations between the artefacts created during the development of software system is essential to several activities of software development such as impact analysis, software maintenance and evolution, component reuse, verification and validation. It is difficult or even impossible to indentify manually these relations in complex systems (e.g. multi-agent systems).

The difficult to indentify traceability relations in multi-agent systems are due to (a) the large number and heterogeneity of documents generated during the development of these systems, (b) the lack of support for the whole development life-cycle by existing agent-oriented methodologies requiring the use of different methodologies, and (c) the possible incompleteness of the documents and models generated during the development of the systems.

We recognize that the above problems can occur in other types of complex systems, but in this thesis we focus on multi-agent systems developed using BDI architecture. In particular, the main differences are the types of the elements and documents that are used when developing a multi-agent system. The development of multi-agent systems involves a new set of elements such as goals, percepts, beliefs, capabilities, agents, roles, actions, events, messages, and plans. To utilize and understand the traceability relations, it is necessary to define the semantics of the relations between these elements. To address this problem we define a traceability reference model to represent the semantic of traceability relations. The semantic of traceability relation gives the ability to carry out richer kind of analysis (e.g. impact analysis).

Another difference is that in some of methodologies such as Troops and Prometheus the definition of requirements is based on goal oriented techniques instead of textual descriptions that allow the development of multi-agent using a model driven development since the requirement definition phase.

Multi-agent systems are distributed and concurrent, and the agents that make up a multi-agent system are able to exhibit complex flexible behaviour in order to achieve its objectives in the face of a dynamic and uncertain environment. This flexible behaviour is key in making agent technology useful, but it makes it difficult to trace agent systems. Tracing is an essential part of the process of developing software, and important to support verification, validation and debugging.

In order to alleviate the above problems, in this thesis we propose the use of software traceability and identification of missing elements between artefacts produced during the whole life cycle of a multi-agent system.

Software traceability has been defined as "the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)" (Gotel, et al., 1994). Traceability relations can help to assist with several activities during the life cycle of software development such as impact analysis, verification and validation, reuse, and maintenance.

The identification of traceability relations manually is a labour intensive and an error prone task (Spanoudakis, et al., 2005). Several approaches have been proposed to recover traceability relations automatically. The approaches can be classified as (i) formal approaches (Pinheiro, et al., 1996), (ii) process oriented approaches (Castro-Herrera, et al., 2007), (Ravichandar, et al., 2007), (Pohl, 1996), (iii) information retrieval approaches (Zou, et al., 2007), (Poshyvanyk, et al., 2007), (Duan, et al., 2007), (Kritzinger, et al., 2008), (Antoniol, et al., 2002), (Marcus, et al., 2003), (Zou, et al., 2006), (De Lucia, et al., 2007), (De Lucia, et al., 2008), (Lormans, et al., 2006), (Hayes, et al., 2007), (iv) string matching approaches (Fiutem, et al., 1998), (Antoniol, et al., 2001), v) rule base approaches (Spanoudakis, et al., 2004), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009), (Cysneiros, et al., 2003), (Cysneiros, et al., 2007a), (Cysneiros, et al., 2007b), (Cysneiros, et al., 2008) (Spanoudakis, et al., 2003), (Spanoudakis, et al., 2004), (Dagenais, et al., 2007), (Reiss, 2006), (Fletcher, et al., 2007), (Rilling, et al., 2007), (Kagdi, et al., 2007), (Alves-Foss, et al., 2002), (vi) run-time approaches (Liu, et al., 2007), (Egyed, 2003), (Egyed, et al., 2005), (Grechanik, et al., 2007), and (vii) hypermedia and information integration approaches (Sherba, et al., 2003), (Sherba, 2005).

The approaches above address different aspects of the traceability problem. For instance, i) formal approaches can be used when it is possible to define the software project using a formal language and then traceability relations are derived automatically using axioms; ii) process oriented approaches can be used when a unified software process development is used to develop software; iii) information retrieval techniques have been used successful to identify traceability relations between textual documentation of software artefacts; iv) string matching approaches can be used when naming of elements are used consistently to define elements of a software project; v) rule-based approaches can be used when it is easy to identify and define rules between relations of elements created during the development of a software; vi) run-time approaches can be used when code of the system is available.

In this thesis a rule-based framework is described to support automatic generation of traceability relations and identification of missing elements in artefacts created during the development of multi-agent systems. The identification of missing elements is called *completeness checking* in this thesis report. This work provides support for artefacts created during different phases of the software development life-cycle. More specifically, the approach supports artefacts created during early and late requirements elicitation, analysis and design, and implementation phases of the development of multi-agent systems.

The framework concentrates on early requirements represented using *i** framework (Yu, 1995), late requirements, analysis and design specification created using the Prometheus methodology (Padgham, et al., 2004), and code implemented with JACK (Agent Oriented Software Limited, 2010).

Prometheus methodology was chosen because it has been largely used in academic and industrial settings; it covers the whole life cycle of development; and there are a large number of documentation, examples, and tools support available. Moreover, the detailed design phase of Prometheus covers the concepts necessary to model multi-agent systems implemented using the BDI architecture.

The reason for using *i** framework to represent the early phase of the requirement specification is due to the fact that Prometheus methodology only supports a specification of goals of the system in terms of a hierarchical diagram. The *i** framework provides a richer modelling technique to represent organizational process. The *i** framework represents relations between actors that depend of each other to have its goals accomplished. The rationale behind the dependencies can also be represented in *i**.

The adoption of JACK is due to its use in several academic and commercial applications and in diverse areas such as unmanned aerial vehicles, surveillance, air traffic management, real-time scheduling, and virtual actors (Agent Oriented Software Limited, 2010). Moreover, a large number of documentation is available and the detailed phase of Prometheus describes the elements of the JACK.

In order to support the heterogeneity of models and tools covered in this work, the models are represented in XML (XML, 2010). XML was chosen as the basis of our approach due to several reasons: (a) XML has become the de facto language to support data interchange among heterogeneous tools and applications, (b) the existence of large number of applications that use XML to represent information internally or as a standard export format, and (c) to allow the use of XQuery (XQuery, 2010) as a standard way of expressing traceability rules.

We propose to use an extended version of XQuery to represent the rules in our framework. XQuery is an XML-based query language that has been widely used for manipulating, retrieving, and interpreting information from XML documents. Apart from the embedded functions offered by XQuery, it is possible to add new functions. We have extended XQuery (a) to support representation of the consequence part of the rules, i.e. the actions to be taken when the conditions are satisfied, and (b) to support extra functions to cover (i) some of the traceability relations being proposed and (ii) completeness checking of the models.

A prototype tool has been implemented to demonstrate and evaluate the work. The evaluation of the work has been performed in three case studies, namely: (i) automatic teller machine, (ii) electronic bookstore, and (iii) air traffic control environment. The automatic teller machine is a small size application that allows a customer to withdraw cash and print statements. The air traffic environment is a medium size application that simulates the landing sequencing of an aircraft. The electronic bookstore application is a large size application that implements the main functionalities of an electronic bookstore such as browsing catalogue, search books by keyword and buy a book.

The remainder of this chapter describes the hypotheses, problem definition, objectives, contributions, and thesis outline.

1.2 Hypotheses

The hypothesis of our framework consists on the identification of traceability relations between software artefacts created during the development of multi-agent systems using a model driven approach. This hypothesis is broken into the following:

- It is possible to use rules to identify traceability relations between software artefacts created during the development of multi-agent systems using a model driven approach;
- It is possible to use rules to identify missing elements between software artefacts created during the development of multi-agent systems using a model driven approach;
- It is possible to use the information about missing elements to fix discrepancies between names given to elements in the different documentation and to improve completeness between software artefacts created during the development of multi-agent systems using a model driven approach;
- It is possible to use the information about missing elements to improve the number of traceability relations identified by our framework.

To evaluate this hypothesis, a prototype tool was built and assessed in three case studies. In chapter 5 these experiments are described and the results of the evaluation presented.

1.3 Objectives

The overall aim of this research is to develop an approach to support traceability between artefacts created during the entire life cycle of the development of a multi-agent system. In particular, the main interest was in supporting the identification of missing elements and automatic generation of traceability relations between software elements in *i** organisational models (Yu, 1995), Prometheus models (Padgham, et al., 2004), and JACK code (Agent Oriented Software Limited, 2010).

The main aim was broken down into the following objectives:

- To define different types of traceability relations;
- To create a reference model that defines traceability relations between artefacts in *i** and Prometheus and between artefacts in Prometheus and JACK code;
- To create a set of rules to identify missing elements and traceability relations between *i** and Prometheus artefacts;
- To create a set of rules to identify missing elements and traceability relations between Prometheus and JACK code elements;
- To develop a prototype tool to identify missing elements and to automatically generate traceability relations between *i** and Prometheus models and between Prometheus models and JACK code;
- To evaluate the work in several case studies.

1.4 Contributions

This research contributes to the Agent Oriented Software Engineering area and addresses the problems discussed in Section 1.1. The main contributions can be summarised as:

- Automatically recovery of traceability relations - A rule-based approach was proposed to support automatic generation of traceability relations between heterogeneous software models created during the development of multi-agent systems. This alleviates the problems of creating traceability relations manually;
- Support for completeness checking - A rule-based approach was proposed to support the identification of missing elements in various software models. This facilitates fixing inconsistencies among the models;
- Traceability Reference Model - Nine types of traceability relations with different semantics and a traceability reference model between *i** and Prometheus elements and between Prometheus and JACK elements were proposed;

- Rules to recover traceability relations and to identify missing elements - Several rules were created to identify missing elements and to recover traceability relations between *i** and Prometheus and between Prometheus and JACK elements;

- Traceability prototype tool - A prototype tool has been developed in order to execute the rules and to create traceability relations and identify missing elements information;

- Development of three case studies – The work was evaluated in three case studies. A small size application of an Automatic Teller Machine where JACK code provided by AOS (Agent Oriented Software Limited, 2010) has been reversed engineering to create Prometheus model. This work shows that the prototype tool can identify automatically most of traceability relations correctly and we used the information about missing elements to fix the inconsistencies and to complete the models. A medium size application of an Air Traffic Control Environment where JACK code has been provided and it was used to create models in Prometheus and *i**. A large size case study of an Electronic Bookstore. In this case, Prometheus models have been created based on available documentation (Padgham, et al., 2004) and on real applications such as Amazon.com (Amazon.com, 2010). JACK code was implemented based on the created Prometheus models.

1.5 Thesis Outline

The remaining of this thesis is structured as follows. In chapter 2 the literature about traceability is reviewed. We introduce what traceability is and the importance of traceability in the software development process. This chapter describes the main traceability reference models, approaches used to recover traceability relations, approaches to represent and maintain traceability relations, approaches to use and visualise traceability relations, and approaches that define traceability reference models. We also describe existing work on traceability and agent oriented systems.

A traceability reference model for software models created during the development of multi-agent systems using *i** framework, Prometheus methodology and JACK language is presented in chapter 3. We describe the elements of *i** framework, Prometheus methodology, and JACK language used in our framework, and nine types of traceability relations.

The framework is described in chapter 4. Initially, we give an overview of the approach and then we provide details of the architecture of the framework. We show how different types of rules can be created to be used by the framework and then give some examples of different type of rules. We describe different functions that we have developed to support the rules, to perform completeness checking, to verify if names of elements in the models are synonyms, to compare similarities between elements in the models, and to manipulate elements in PDT (PDT, 2010), TAOME (TAOME4E, 2008) and JACK models. Finally, we describe the prototype tool that we have developed to support our traceability framework.

The evaluation of the framework in three case studies and the results of this evaluation are presented in chapter 5.

In the chapter 6 the conclusions and future works are presented. The main contributions of the research and how the hypotheses have been achieved are described.

The thesis report is composed of several appendices. Appendix A contains the list of extra functions implemented in Java to extend XQuery. Appendix B describes the Automatic Teller Machine case study. Appendix C describes the Air Traffic Control Environment case study. Appendix D describes the Electronic Bookstore case study. Appendix E gives an introduction to the BDI architecture. We present different types of agent architecture used to build multi-agents systems and then we describe in detail the BDI architecture that was used by our research. Appendix F describes traceability relations between *i** and Prometheus elements. Appendix G describes traceability relations between Prometheus and JACK elements.

Chapter 2 - Literature Survey on Traceability

Software traceability is the ability to relate artefacts created during the development life-cycle of a software system (Spanoudakis, et al., 2005). More specifically, from the point of view of requirements, traceability has been defined as the "*ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)"* (Gotel, et al., 1994).

Software traceability is essential in the software development process and has been used to support several activities such as impact analysis, software maintenance and evolution, component reuse, verification, and validation. The importance of traceability in the software development process has been endorsed by several standards for quality management and process improvement such as ISO 9001:2000 (ISO, 2010) and CMMI (Carnegie Mellon, 2010).

Gotel discusses several challenges and problems (Gotel, 2008), (Gotel, 2009) that exist to support traceability practice. Examples of these problems and challenges are: i) traceability is seen as a repetitive and tedious task and the challenge is how to change this image (*the yo-yo challenge – the boredom of a fixed routine*); ii) responsibility for traceability is blurred and the challenge is to distribute the responsibility for traceability to all team members of a software project ; iii) artefacts in the software project are from different types and they are represented in a variety of medias, and at different levels of formality and granularity and the challenge is to identify what should be traced and how trace relations are to be established; iv) the credibility of traceability can be debatable and the challenge is how determine ways to communicate confidence level of trace relations; v) traceability relations tend to *decay* without dedicated ongoing maintenance and the challenge is to plan a traceability strategy well; vi) unrealistic expectations are placed on traceability automation, however techniques to recover traceability relations automatically still demand a high quality set of artefacts and manual filtering of results and the challenge is to figure out how to combine heterogeneous automated and humans approaches to support traceability; vii) traceability should be a by product, but it became an extra activity of software development and the challenge is to make traceability to be achieved as a by-product of other engineering activities.

In addition, several researchers and practitioners have participated in a series of two events: First Workshop on Grand Challenges for Traceability (GCW'06) and International Symposium on Grand Challenges in Traceability (GCT'07) with the goal of identifying challenges in the area of traceability that need to be addressed. The identified challenges in these workshops have led to the creation of the Grand Challenge document. Examples of these identified challenges are: Traceability Knowledge, Training and Certification, Supporting Evolution, Link Semantics, Scalability, Human Factors, Cost Benefit Analysis, Methods and Tools, Tracing across Organizational Boundaries, Process, Compliance, Measurements and Benchmarks, Technology Transfer (Cleland-Huang, et al., 2007).

Despite the importance of software traceability, current support for traceability is inadequate. Most of the commercial tools do not provide mechanisms to automatically generate and maintain traceability relations. Moreover, existing tools do not offer support for defining the various types of the traceability relations (i.e., the semantics of the relations). The lack of automation becomes a serious problem in the development of complex software systems where the numbers of artefacts are large and there is a need to establish traceability relations between those artefacts that are usually created by non-interoperable tools, and can evolve autonomously.

De Lucia et al. (De Lucia, et al., 2007) discuss the importance to show the effectiveness of traceability recovery approaches. In particular, they present a study that compares the effort to identify traceability relations using a traceability tool (i.e. ADAMS) with the effort to identify traceability relations manually. As it is expected, the study shows that the use of a tool helps to improve precision and it also reduces the time necessary to indentify traceability relations. Similarly, Grechanik et al. (Grechanik, et al., 2007) shows that less time was spent by software analysts when using their approach to automatically identify traceability relations in order to execute task of code evolution and code comprehension.

In (Asuncion, 2007), the authors declare that the effective practice of traceability aids in system comprehension, impact analysis, system debugging, and communication between the development team and stakeholders. Assunction et al. present the economic, technical and social benefits obtained using a traceability tool that supports the entire life cycle of the software development in an industrial case study. For instance, some benefits derived from the traceability practice are: i) raise of the visibility of actual software processes enabling users to compare actual practices to stated company procedures; ii) automation replace burdensome tasks associated with traceability, such as maintaining consistency between various artefact representations; iii) prove to the customers that the requirement has been tested; iv) project managers easily obtain an accurate status report of the project.

Traceability has been studied for many years and several approaches have been proposed to tackle its different aspects and issues. Pohl (Pohl, 1996) states that a traceability approach should provide answers to the following questions:

- What traceability information should be captured?
- How traceability information should be captured?
- How traceability information should be stored?

Sherba adds in (Sherba, 2005) that a traceability approach should also to answer the question below:

- How traceability relations are going to be viewed and queried?

Moreover, Gotel (Gotel, 2009) states that before trying to answer what, where, when, and how to trace, it is necessary to answer "why it is important to trace?". Gotel highlights that it is important to know who are the stakeholders and what are their needs. Gotel says that traceability is a team effort and that other stakeholders need to understand why they should spend time to create or to maintain traceability relations to other stakeholders. Gotel et al. also discuss (Gotel, et al., 2007) that some lessons can be learned from other industries that use traceability. In particular, Gotel et al. compare traceability in software industry with traceability in the food industry. For instance, the responsibility for traceability in the food industry is shared by all people involved in a certain process, while the responsibility for traceability in the software industry is assigned to a few people in most of the cases.

In the next sections, we discuss how different approaches address the questions above and provide a literature survey of these existing approaches.

2.1 Traceability Reference Models and Meta-Models

Traceability reference models are used to define the types of traceability information that should be captured. Several classifications for different types of traceability relations and several traceability reference models and meta-models have been proposed in the literature (Davis, 1990), (Gotel, et al., 1994), (Lindvall, et al., 1996), (Dick, 2002), (Ramesh, et al., 2001), (Spanoudakis, et al., 2005), (Berenbach, 2007), (Almeida, et al., 2007), (Goknil, et al., 2008), (Toranzo, et al., 1999), (Toranzo, et al., 2002), (Han, 2001), (Pinto, et al., 2005).

In (Davis, 1990), traceability has been classified from the perspective of direction as *forward* and *backward*. Forward traceability is the ability to trace an artefact to its implementation, while backward traceability is the ability to trace an artefact to its origin. More specifically, Davis has identified four types of traceability relations, namely (a) forward from requirements, (b) backward to requirements, (c) forward to requirements, and (d) backward from requirements. Types (a) and (b) are also known as post-traceability and types (c) and (d) are known as pre-traceability (Gotel, et al., 1994). Traceability relations can also be categorised as horizontal (or inter-traceability) and vertical (or extra-traceability) (Lindvall, et al., 1996). Horizontal traceability relations refer to those relations within the same model, while vertical traceability relations refer to those relations that involve different models.

The need to capture the semantic of traceability relations has been point out as fundamental in order to make effective the use of traceability (Dick, 2002), (Ramesh, et al., 2001), (Spanoudakis, et al., 2005). Dick proposed in (Dick, 2002) an approach to represent "deeper kinds of traceability" relations in order to perform deeper types of analysis. He argues that the use of propositional logic to group relations together and textual information describing the rationale of the relations can be used to describe the traceability relations and to perform further analysis.

Ramesh et al. describe two traceability reference models in (Ramesh, et al., 2001). These traceability reference models have been derived from an empirical study of traceability practices in 26 major software development organizations. In this study, they identified two types of traceability users: "low-end" and "high-end" users. Low-end users have few years of experience with traceability and the use of traceability is compelled by project sponsors or for compliance with standards. Low-end users simply use traceability to relate various components of information without explicit identification of the semantic and rationale of such relations. The main application of traceability by low-end users is for requirements decomposition, requirements allocation, compliance verification, and change control. On the other hand, high-end users have several years of experience and use traceability to cover the full cycle of the software development, and to capture discussion issues, decisions, and rationale.

Low-end users use traceability to create relation of: i) dependencies between requirements (*derive)*; ii) allocation of requirements to system components (*allocated_to*); iii) satisfaction of requirements to system components (*satisfy*); iv) compliance verification procedure developed for requirements (*developed_for*); v) dependencies between system components (*depend_on*); vi) compliance verification performed by system components (*performed_*on); vii) between interfaces that system components has with external systems (*interface_with*). High-end users use much richer traceability schemes than low-end users. Ramesh et al. divided traceability relations in four parts: Requirements Management, Design Allocation, ComplianceVerification, and Rationale Management.

In (Berenbach, 2007), Berenbach proposes a traceability meta-model and affirms that the implementation and tool support of traceability can help to enforce design and process rules. Berenbach says that a traceability meta-model can be used to create completeness verification checks, and that traceability information can be used to propagate name changes of related elements. The reference model proposed by Berenbach consists of traceability relations between elements of analysis and design models in UML. Examples of types of relations are: i) requirements derive from (*Derive From*) use cases; ii) use cases are associated with (*associated with*) use cases; iii) use cases are shown on (*shown on*) use case diagrams; iv) use case realizations implements (*realize*) use cases; vi) use cases are explained by (*explained with*) sequence diagrams; vii) business object relationships are shown on (*Relationships shown on*) class diagram; viii) boundary elements interact with business objects; ix) use case realization are contained in (*contained in*) design package; x) components realizes (*realizes*) business object; xi) component has (*has*) interface; xii) component are shown on (*shown on*) component diagram; xiii) components are composed of (*composed of*) classes; xiv) classes relations are shown (*relationships shown on*) on class diagrams; xv) test case tests (*tests*) components; xvi) test case verifies (*verifies implementation*) requirement; xvii) subsystem details are shown in (*details shown in*) sequence diagrams; xviii) subsystem behaviour are explained by (*behaviour explained by*) activity diagram, subsystem contains (*contains*) design package.

Almeida (Almeida, et al., 2007) proposes a requirement traceability meta-model to support Model Driven Engineering development. The meta-model is implemented using Ecore metamodel. The meta-model proposed by Almeida et al. consists of satisfaction traceability relations between requirements that are part of the requirement specification and artefacts of the software model.

Goknil et al. highlight the importance to define the semantics of traceability relations in order to execute change and impact analysis activities (Goknil, et al., 2008). Goknil presents a traceability meta-model composed of four different types of traceability relations between requirements, namely: requires, refines, conflicts, and contains. A requirement R_1 requires a requirement R_2, if R_1 is fulfilled only when R_2 is fulfilled. A requirement R_1 refines a requirement R_2, if R_1 is derived from R_2 by adding more details to it. A requirement R_1 contains requirements R_2 if R_2 is part of the requirements R_1. A requirement R_1 conflicts with a requirement R_2, if the fulfilment of R_1 excludes the fulfilment of R_2 and vice versa.

Another traceability reference model for Model Driven Engineering development has been proposed in (Vanhooff, et al., 2005). Vanhooff defines traceability information used by model transformation as *transformation traceability*. Vanhooff presents a traceability meta-model that consist of dependencies between source and target elements, dependencies between a mapping and the transformation unit that created it, and the marking of source element as deleted.

Toranzo et al. (Toranzo, et al., 1999), (Toranzo, et al., 2002) present a general purpose reference model. Examples of relations are: i) stakeholders are responsible for (*responsibility*) requirements, a program represents (*represents*) requirements, requirements are allocated to (*allocated_to*) sub-systems, and tasks are satisfied by (*satisfy*) design elements. Pinto et al. (Pinto, et al., 2005) describe a process for guiding the use of a reference model to the development of multi-agent systems. In particular, Pinto et al. propose a series of guidelines to extend Tropos in order to support traceability.

Han (Han, 2001) describes a traceability model between requirements and architecture documents. Examples of types of relations are: i) components provides (*provides*) services; ii) components requires (*requires*) services; iii) components are part of (*part_of*) components; iv) components conforms to (*conforms_to*) interface; v) interface makes visible (*makes_visible*) assumptions; vi) assumptions are subject to (*subject_to*) risks; vii) authorities asserts (*asserts*) assumption; viii) services respect (*respect*) assumption; ix) stakeholders owns (*owns*) goals; goals refines (*refines*) goals; x) services are delivered with (*derived_with*) quality of services; xi) quality of services satisfies (*satisfies*) goals; xii) use cases uses (*uses*) services.

In (Pohl, 1996), the authors identify 18 types of traceability relations that were created based on a survey of the requirements engineering literature. The types of traceability relations were created to describe relations between a hypertext model that specify the vision and requirements of the system, a structured analysis (SA) model that consists of a data flow model and a data dictionary, an extended entity-relationship (ER) model for modelling the data view of the system and an OMT model for modelling the object-oriented view as well as the behaviour of the system.

The traceability relations are classified in five groups, namely: (a) condition links, (b) context link, (c) document links, (d) evolutionary links, and (e) abstraction links. Condition links are used to relate restrictions (precondition or constraints) to a particular object; context links are used to express relations of similarity, comparisons, contradictions, and conflicts between objects; document links are used to relate different kinds of documentation to a requirement such as examples, test cases, description of the purpose, background information and comments; evolutionary links are used to express when a requirement has been replaced, based on, formalized or elaborated by another requirement; abstraction links are used to represent abstractions between trace requirements such as generalizations or refinements.

Spanoudakis and Zisman (Spanoudakis, et al., 2005) propose a framework to organize the types of traceability relations identified in the literature. They had grouped the traceability relations into eight main groups: i) dependency; ii) generalisation/refinement; iii) evolution; iv) satisfaction; v) overlap; vi) conflicting, vii) rationalisation; viii)contribution. Dependency relations are used if an element relies on the existence of another element, generalization/dependencies are used to identify how complex elements of a system can be divided into components, or how an element can be specialised by other elements or how elements can be generalised by another element, evolution relations are used if an element is replaced by another element, satisfiability relations are used if an element meets the expectation, needs and desires of another element or if an element complies with a condition represented by another element, overlaps relations are used if two elements refer to common features of a system or its domain, conflict relations are used to represent conflicts and issues between two elements, rationalisation are used to represent and maintain the rationale behind the creation and evolution of elements, and decisions about the system at different levels of detail, contribution relation are used to represent associations between requirement artefacts and stakeholders that have contributed to the generation of the requirements.

Spanoudakis et al. describe in (Spanoudakis, et al., 2004), a rule based approach to identify traceability relations between requirements specifications using structured text and use case specifications using Cockburn template (Cockburn, 2000), and between requirements and class diagrams. The approach defines four different semantic types of traceability relations, namely: overlaps, requires_execution_of, requires_feature_in, and can_partially_realise. An overlaps relation is used if two elements refer to a common feature of the system or its domain; a requires execution of (*requires_execution_of*) is used when a sequence of terms appears in a pre-condition of an use case, post-condition of an use case, a requirement statement or an use case event requires the execution of an operation. A requires feature (*requires_feature_in*) relation is used between a part of an use case specification and a requirement statement (r_2), or between a requirement statements r_1 and another requeriment statement r_2 when the use case or the requirement cannot be realised without the existence of the structural or functional feature in the requirement r_2. A can partially realise (*can_partially_realise*) relation is used between a description, an event or a postcondition of a use case and the description of a requirement statement if the use case can realise part of the requirement statement.

An extension of the above work has been proposed by Jirapanthong et al. in (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009). The work in (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009) identifies traceability relations between documents created to develop software product families. The approach helps to identify common and variable aspects between different members of a product family. The approach identifies traceability relations between different types of documents generated when using an extension of the FORM methodology to develop product family systems. A traceability reference model has been created and nine types of traceability relations are proposed: satisfiability, dependency, overlaps, evolution, implements, refinement, containment, similar, and different. Satisfiability relations are used if an element meets the expectations and needs of another element. Dependency relations are used if the existence of an element relies on the existence of another element. Overlaps relations are used if two elements refer to common aspects of a system or its domain. Evolution relations are used if an element has been replaced by another element. Implement relations are used if an element executes or allows for the achievement of another element. Refinement relations are used to identify how complex elements can be decomposed in sub-elements. Containment relations are used when an element uses another element. Similar relations exist between elements that depend on the existence of a relation in common. Different relations are used to assist with the identification of variable aspects between various product members.

Although the reference models and types of relations presented in the literature provide a better understanding of the semantics of traceability relations, there is no consensus on the different types of traceability relations (Sherba, 2005). Moreover, the types of traceability relations are project specific (Pinheiro, et al., 1996), (Spanoudakis, et al., 2005) and can vary depending on the stakeholders, methodologies, domain, and tools involved in the system software development process. Therefore, it is important to create an approach that allows stakeholders to define the type of traceability relations that are important to them in a particular project.

Moreover, to the best of our knowledge, there are no traceability reference models for elements created during the development of multi-agent systems using *i** framework, Prometheus methodology and JACK language. The granularity of traceability relations created in (Pinto, et al., 2005) is a general traceability reference model for elements created during the development of multi-agent systems and it has been created to enhance the Tropos methodology to support traceability. The granularity of the types of relations between code elements (i.e. "Program") and design elements are high level and it does not take in consideration different types of code elements and design elements.

2.2 Traceability Approaches to Capture Trace Relations

There are several types of tools that provide support for capturing traceability in various activities of the software development life-cycle. Examples of these tools are requirement management tools, software change and configuration management tools, and project management tools. However, most of these tools require some intervention by the user in order to create traceability relations. Moreover, in these types of tools, the user has to select the source and target elements to be related. Some of these tools provide some mechanism to assist with the definition of traceability relations. For example, Rational DOORS (IBM Rational, 2010a) and CaliberRM (Borland, 2010) can import the requirements automatically from documents in Microsoft Word based on the heading styles of the text, while Rational RequisitePro (IBM Rational, 2010b) can import the requirements based on keywords. However, once the requirements have been imported, the relations have to be identified manually.

The evidence of the importance of tools to support traceability is the large number of commercial tools available in the market (Standish Group, 2003). Examples of requirement management tools are Rational DOORS, Rational RequisitePro, and CaliberRM. Most of commercial tools available to support traceability require the user define traceability relations manually or provide limited support to automatic creation of the traceability relations.

The task of creating traceability relations manually is costly, labour-intensive, and error-prone (Spanoudakis, et al., 2005), (De Lucia, et al., 2008), (Hayes, et al., 2004), (Lormans, et al., 2006). As a consequence, the cost to establish traceability relations can overcome its benefits. To address this problem several approaches have been proposed to support automatic creation of traceability relations. We classify these approaches in seven groups based on the techniques that they use to support generation of traceability relation, namely (i) formal approaches (Pinheiro, et al., 1996); ii) process oriented approaches (Castro-Herrera, et al., 2007), (Ravichandar, et al., 2007), (Pohl, 1996); iii) information retrieval approaches (Zou, et al., 2007), (Poshyvanyk, et al., 2007), (Duan, et al., 2007), (Kritzinger, et al., 2008), (Antoniol, et al., 2002), (Marcus, et al., 2003), (Zou, et al., 2006), (De Lucia, et al., 2007), (De Lucia, et al., 2008), (Lormans, et al., 2006), (Hayes, et al., 2007); iv) string matching approaches (Fiutem, et al., 1998), (Antoniol, et al., 2001); v) rule base approaches (Spanoudakis, et al., 2004), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009), (Cysneiros, et al., 2003), (Cysneiros, et al., 2007a), (Cysneiros, et al., 2007b), (Cysneiros, et al., 2008) (Spanoudakis, et al., 2003), (Spanoudakis, et al., 2004), (Dagenais, et al., 2007), (Reiss, 2006), (Fletcher, et al., 2007), (Rilling, et al., 2007), (Kagdi, et al., 2007), (Alves-Foss, et al., 2002); vi) run-time approaches (Liu, et al., 2007), (Egyed, 2003), (Egyed, et al., 2005), (Grechanik, et al., 2007); vii) hypermedia and information integration approaches (Sherba, et al., 2003), (Sherba, 2005).

2.2.1 Formal Approaches

Formal approaches define software artefacts and their relations using a formal language, and by using axioms and regular expression to identify traceability relations between the artefacts (Pinheiro, et al., 1996). The main problem when using these formal approaches is the need to have training and knowledge of a specific formal language. To alleviate this problem TOOR approach (Pinheiro, et al., 1996) uses graphical interface where the specification of the project and the relation between the artefacts can be defined using a combination of graphical interface and forms.

TOOR (Pinheiro, et al., 1996) provides a semi-automatic approach to identify traceability relations and the process of capture traceability relations is divided in three different phases: definition, registration, and extraction. In the definition phase, the user defines the classes of objects and types of relations to a specific project using the FOOPS (Functional and Object-Oriented Programming Systems) formal language. In the registration phase, the objects are created by the selection of the appropriate class of the object from a graphical user interface and then a template form is filled. To create a traceability relation, the user has to select the type of traceability relation and fill a template form with the source and the target object from the relation. Alternatively, a graphical user interface can be used to select the source and target objects. Relations can also be created based on axioms defined in the definition phase. Finally, in the extraction phase, the axioms are computed and the traceability relations are displayed.

2.2.2 Process Oriented Approaches

Traceability is required by several standards for quality management and process improvements such as ISO 9001:2000 (ISO, 2010) and CMMI (Carnegie Mellon, 2010). Some approaches integrate traceability techniques with software process (Castro-Herrera, et al., 2007), (Ravichandar, et al., 2007), (Pohl, 1996). The main advantages of these approaches are that traceability relations are created as a product of the software development process and also enforce a practice in the software development process. The main disadvantages of these approaches are concerned with the difficulty to support tool integration and the lack of the definition of unified software development processes in practice.

Castro-Herrera et al. (Castro-Herrera, et al., 2007) propose to extend Basic Open Unified Process (OUP/Basic) to incorporating automated traceability adding new documents (i.e. *work products*) and tasks to the process. The authors highlight the importance and the need to integrate automated traceability techniques in the software development process to maximize the potential benefit of automated traceability. Castro-Herrera added three new work products: Requirements document, Trace Strategy and Granularity document, and Additional Traceable document and five new tasks: Create Trace Strategy, Create Additional Traceable documents, Set Up in Place Traceability, Run Automated Traceability Analysis, and Test and Verify Automated Traceability Results. Requirements documents describe the functional requirement using a textual description. Trace Strategy and Granularity document is used to describe different traces that the stakeholders wish to record. Additional Traceable document is a general template that can be instantiated to include other artefacts that are not defined in the process. Create trace strategy task defines the traceability strategy and granularity of the traceability relations. Create additional traceable documents task creates new artefacts that need to be traced based on some guidelines. Set up in place traceability task set up the infrastructure necessary for the use of traceability tools. Run automated traceability analysis task executes the automated traceability tool and provides feedback. Test and verify automated traceability results task analyses the effectiveness of automated traceability.

Pohl (Pohl, 1996) presents a process centred approach that automatically creates the traceability relations by recording the execution of actions during the software system development. The approach requires a method engineer to define the process and tools that are stored in a repository. Traceability relations are also identified automatically as part of the software process in (Ravichandar, et al., 2007). Ravichandar et al. uses Capability Engineering process to identify systems requirements from user needs. A graph (Function Decomposition) is created to link the decomposition between different levels of abstraction of user needs.

Traceability relations can be inferred by the transformations from the user needs to the requirements represented in the Function Decomposition graph.

2.2.3 Information Retrieval Approaches

Several approaches use information retrieval techniques to identify traceability relations between software artefacts (Zou, et al., 2007), (Poshyvanyk, et al., 2007), (Duan, et al., 2007), (Kritzinger, et al., 2008), (Antoniol, et al., 2002), (Marcus, et al., 2003), (Zou, et al., 2006), (De Lucia, et al., 2007), (De Lucia, et al., 2008), (Lormans, et al., 2006), (Hayes, et al., 2007). Information retrieval techniques identify traceability relations based on the fact that artefacts with high textual similarities probably share concepts and, therefore, are likely candidates to have traceability relations. The main drawbacks of using information retrieval techniques to identify traceability relations is that standard information retrieval techniques do not take into consideration the structure of the artefacts. Moreover, in these approaches, a large percentage of candidate relations are identified (high recall), however the percentage of identified candidate relations that are correct is low (low precision) (Zou, et al., 2007). This increases the effort necessary to select from the set of candidate relations what relations are correct and what relations are invalid. Some approaches address this problem mainly by incorporating coupling techniques (Poshyvanyk, et al., 2007), clustering methods (Duan, et al., 2007), phrasing (Zou, et al., 2007), query term coverage (Zou, et al., 2007), relevance feedback, and attribute weighting (Kritzinger, et al., 2008) to the information retrieval technique.

Antonio et al. (Antoniol, et al., 2002) describe an approach to identify traceability relations between source code and natural language documentation based on information retrieval techniques using both a probabilistic method and vector space model. The approach uses comments and identifier names within the source code to find similarities in the documentation. The documents are ranked by relevance and based on these relevance the traceability relations are created.

Another approach named Latent Semantic Indexing (LSI) is described in (Marcus, et al., 2003). Marcus et al. argue that their approach achieves better results than the Antoniol's approach. Their approach uses full parsing code and morphological analysis of the documentation Marcus et al. affirm that, in comparison with Antoniol's approach (Marcus, et al., 2003), their approach requires less processing of the source code and documentation, and it is language, programming language, and paradigm independent.

Poshyvanyk et al. presents (Poshyvanyk, et al., 2007) an approach that combines LSI technique to recover traceability relations between software documentation (e.g. requirements and user manuals) and code with coupling measures techniques. The main goal of the approach is to address the common problem that the structure of the documentation (e.g. files, sections of documents, directories, etc.) does not reflect the structure of the source code.

The use of clustering to reduce effort required by the user to select candidate relations generated by information retrieval techniques is investigated by Duan et al. in (Duan, et al., 2007). Three algorithms have been implemented and tested using a web-based tool named Poirot. The tested algorithms are agglomerative hierarchical clustering, K-means, and bisecting divisive clustering and they were evaluated to capture traceability relations between requirements and other types of artefacts such as higher level business goal, design elements and code. Duan et al. affirm that the benefit of using the tool was that traceability relations are presented to software analysts as part of a meaningful group (cluster). This allows the analyst to take decision about accept or reject candidates relations based on similar artefacts. The tool also provides functionality that allows the user to accept or reject all candidate relations associated with a specific cluster.

Zou et al. declares in (Zou, et al., 2007) that most of the information retrieval techniques used to recover traceability relations are able to find a large percentage of correct relations (high recall), but in general produce a low level of precision. To address this problem Zou et al. propose the use of phrasing in (Zou, et al., 2006). Information retrieval techniques such as vector space model, probabilistic model, and latent semantic indexing build an index of terms used by the documents. Zou et al. approach uses phrases instead of single terms. They assume that artefacts that share common phrases are more inclined to be related than artefacts that only share common terms. The phrases are automatically generated by the tool making use of a part-of-speech tagger and searching the entire document. The approach is extended in (Zou, et al., 2007) to use query term coverage. Query term coverage takes into consideration the number of unique shared terms, while in standard information retrieval techniques the similarity is based on the total weight that is calculated based on the frequency in which the terms appear in a document.

Kritzinger et al. propose an approach (Kritzinger, et al., 2008) that uses latent semantic analysis to identify traceability relations between several software artefacts such as system requirements, use cases, collaboration and state diagrams, and source code in C#. The Kritzinger's approach differs from others approaches that use latent semantic analysis mainly by incorporating user relevance feedback and attribute weighting to the technique. The user feedback helps to create cluster of documents that are relevant to previous queries. Attribute weighting takes into consideration the structure of a document (e.g. methods, fields and package declarations of a class) during term weighting phase to create a term-artefact matrix.

De Lucia et al. (De Lucia, et al., 2008) present ADAMS Re-Trace tool that also uses latent semantic indexing technique to identify traceability relations between artefacts of different types. ADAMS Re-Trace is integrated to Advanced Artefact Management System (ADAMS) that is a fine-grained artefact management system for Eclipse. Traceability relations between artefacts are created manually in ADAMS and used for impact analysis and change management tasks. ADAMS Re-Trace adds to ADAMS system the functionality to identify traceability relations semi-automatically.

Lormans and Deursen also apply Latent Semantic Indexing (LSI) technique in three different case studies to recover traceability relations between requirements and design documents and between requirements and test documents (Lormans, et al., 2006). The cases studies are different in size and scope varying from small to complex, and from academic to industrial. The authors highlight the importance of having a traceability model as part of the traceability approach and classify them in static and dynamic models. In a dynamic model, the types of relations can change according to specific project needs. A static approach is used to define traceability relations types between requirements and design and requirements and test documents. The approach uses a Text to Matrix Generator tool to pre-process documents (e.g. lexical analysis, stop word elimination, stemming, index-term selection and index construction) used as input by the Trace Reconstructor (TR) tool that generates a term-by-document matrix. TR tool selects candidates relations that are greater than a constant value and that have a similarity degree greater than a percentage value calculated based on the total of similarity measures.

Hayes et al. describes in (Hayes, et al., 2007) a front-end for RETRO (Requirements Tracing On target) tool that can be used with different information retrieval techniques. The authors argue that the user satisfaction with a traceability tool depends more on the functionalities provided by the front-end than the information retrieval method used. RETRO has been used in several projects by NASA Independent Verification and Validation (IV & V) Program.

2.2.4 String Matching Approaches

String matching approaches identify traceability relations based on the name of the elements and its properties (Fiutem, et al., 1998), (Antoniol, et al., 2001). Regular expressions, edit distance, and maximum match algorithms (Fiutem, et al., 1998), (Antoniol, et al., 1999) are used in the process of naming comparison. The main drawback of string matching approaches is that they rely on the assumption that artefacts are named consistently through all documentation of a system.

In (Antoniol, et al., 2001), a method to identify traceability relations between object-oriented design and code in C++ is proposed. The method first translates the C++ source code into an intermediate representation and then identifies similarities between the pair of elements from design and code. The similarity comparison is based on matching class names, attributes and their types, and method signatures.

Fiutem et al. (Fiutem, et al., 1998) presents an approach that identifies traceability relations between design elements in OMT and C++ source code elements. The main goal of the approach is to ensure consistency among software artefacts. The approach uses the Abstract Object Language (AOL) to represent design elements in OMT and C++ code. The approach finds traceability relations between elements based on the name of classes and its properties. The approach uses regular expressions and an edit distance algorithm to match the names. The approach provides a visualisation mechanism that show common parts and missing information. Antoniol et al. also use a similar approach in (Antoniol, et al., 1999) to establish traceability relations between different versions of a system implemented using C++ code. The approach provides a visualisation mechanism that compares the difference between two versions. It contains a release view and a class level view. The release view represents graphically the degree of similarity between two versions. The class level view is more detailed and show additions/deletions and modifications of attribute and a file summary.

Our work is similar to string matching approaches in that we also use name of entities and its properties to identify traceability relations. However, our work differs from the string matching approaches on that it uses other information such as traceability relations, to identify similarities between two elements. Moreover, our work uses synonyms to compare similarities between names and the completeness checking supported by our work helps to identify elements that have been named inconsistently (i.e. discrepancy of names between elements), as described in Chapter 4.

2.2.5 Rule Based Approaches

Rule based approaches create traceability relations between elements when a certain condition is satisfied. Examples of rule-based approaches to support generation of traceability relations are (Spanoudakis, et al., 2004), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009), (Cysneiros, et al., 2003), (Cysneiros, et al., 2007a), (Cysneiros, et al., 2007b), (Cysneiros, et al., 2008) (Spanoudakis, et al., 2003), (Spanoudakis, et al., 2004), (Dagenais, et al., 2007), (Reiss, 2006), (Fletcher, et al., 2007), (Rilling, et al., 2007), (Kagdi, et al., 2007), (Alves-Foss, et al., 2002). The main challenge of rule based approaches is make people to understand that is necessary to spend some time a prior to know what need to be traceable and in what ways. Most of the time, it is difficult to identify what is need to be traceable and in what ways and then create rules to identify these relations. To alleviate this problem some traceability approaches create traceability models and pre-established rules to identify traceability relations (Spanoudakis, et al., 2004), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009), (Cysneiros, et al., 2007).

Spanoudakis et al. describe in (Spanoudakis, et al., 2004), a rule based approach to identify traceability relations between requirements specifications using structured text and use case specifications using Cockburn template (Cockburn, 2000), and between requirements and class diagrams. A prototype tool was developed to generate automatically traceability relations. The tool receives as input requirement and use case specifications and a set of rules expressed in

XML. The requirements and use cases specifications are pre-processed by CLAWS part-of-speech tagged tool in order to create a tagged representation of the documents indicating the grammatical role of each word in the text. The documents used by the approach are represented in XML or translated into XML. A machine learning algorithm has been presented in (Spanoudakis, et al., 2003) to improve recall of the approach. The algorithm creates new rules that generalize the syntactic patterns of the original rules based on examples of undetected traceability relations by the user. The traceability rules are defined using XQuery.

Jirapanthong et al. (Jirapanthong, et al., 2005) extends Spanoudakis et al. work to support documents created during the development of product line systems. Jirapanthong et al. approach support different types of documents: feature, subsystem, process, and module models; and class, statechart, and sequence diagrams. The semantic of relations are different in both approaches. Spanoudakis et al. create a XML mark-up language to define rules while in Jirapanthong et al. approach rules are expressed using a XML mark-up language that contains XQuery code segments. Both approaches provide comparison of synonyms in the terms used in the documents. The difference is that in Spanoudakis et al. approach the list of synonyms is created to each project while Jirapanthong and Zisman approach uses WordNet dictionary. The first approach to implement synonym function gives a better precision, since the terms that are considered by the synonym function is specific to the domain or context of the project, but this approach requires an additional effort to create a list of synonyms and the list of synonyms will also not be as complete as a dictionary such as WordNet (WordNet, 2010). The semantic of relations are different in both approaches. Spanoudakis et al. proposes four types of relationships: *overlaps*, *requires_execution_of*, *requires_feature_in* and *can_partially_realise* and Jirapanthong and Zisman propose *satisfiability*, *dependency*, *overlaps*, *evolution*, *implements*, *refinement* and *containment*.

Dagenais et al. presents in (Dagenais, et al., 2007) a plug-in for Eclipse named ISIS4J to infer structural patterns between concerns (e.g. features) and source code. The objective of the tool is to support software evolution. Dagenais et al. say that traceability relations between a concern and code can be invalid or new relations should be created every time that the code changes. To address this problem, Dagenais et al. propose an approach to automatically infer patterns as rules based on a given mapping and a set of pattern rule templates that can be checked as the source code evolves. The paper describes a case study where mapping between concerns and Java code are created using ConcernMapper tool. Checking whether structural patterns hold across different versions of a system enables the automatic identification of new elements related to a document concern and remove elements listed in the concern mapping file that no longer exist and add new elements to the concern mapping file.

The work in (Reiss, 2006) describes a tool to support consistency between different types of artefacts represented in XML, based on the use of constraint rules represented in SQL. The artefacts used in (Reiss, 2006) are mainly UML design models, Java code, and JUnit test package.

The need to verify that a goal has been satisfied in the design model is highlighted by Fletcher et al. (Fletcher, et al., 2007). Fletcher et al. proposes an approach that uses design patterns defined in a Soft Goal Interdependency Graph (SIG) (part of the NFR framework) and UML models. The approach uses a set of rules based on some heuristics to transform goals defined in the Soft Goal Interdependency Graph into a XMI representation that corresponds to elements that are expected to be found in the UML design model. The generated file in XMI is compared with the design model and then traceability relations are created between elements that match them. Missing elements in the design models are reported as broken.

Rilling et al. (Rilling, et al., 2007) argue about the need to make use of the structure and semantic of sentences of textual documents to identify traceability relations. They present an approach where software documents and source code are parsed and stored in ontology. The text parts in the software documents are pre-processed by using tokenisation, sentence splitting, part-of-speech tagging, and noun phrase chunking techniques. These techniques are executed initially, and the resulting texts are used as input, together with some grammar rules, to a syntactic analyser parser. The source code is parsed and then the ontology is populated with both source code and software documents together with their relations. The approach applies text mining to create traceability relations and queries can be executed to find implicit relations between parts of the software document and the source code.

Kagdi et al. (Kagdi, et al., 2007) use some heuristic rules based on patterns of documentation and source code changes to identify traceability relations between source code and other documents such as user documents, build management documents, user guides, release and distribution documentation, and progress reports. A tool called sqminer was developed to identify patterns using sequential pattern-mining techniques. The tool analyse version history of changes of source code and documentation and based on the frequency of occurring co-changes the traceability relations between the documents are created.

Alves-Foss et al. (Alves-Foss, et al., 2002) use Extensible Stylesheet Language (XSL) to create rules to identify traceability relations between UML design elements represented in XML Metadata Interchange (XMI) and Java code represented using JavaML (i.e. a marked up language to represent Java source-code using XML). The traceability relations are represented using XML Linking Language (XLink). The traceability relations are identified based on the name given to the classes.

2.2.6 Run-time approaches

We define run-time approaches as the approaches that uses run-time information of what part of the code has been executed with use case (or scenarios) and a set of initial traceability relations to identify other traceability relations. Examples of rule-based approaches to support generation of traceability relations are (Liu, et al., 2007), (Egyed, 2003), and (Egyed, et al., 2005).

The main drawback of run-time approaches is that they deal with establishing traceability relations after the fact, and not during the creation of artefacts, therefore they are only useful for software maintenance. Moreover, run-time approaches need a set of traceability relations defined manually initially.

Liu et al. shows an approach to identify parts of code that are related to a feature (Liu, et al., 2007). Liu et al. uses a hybrid approach combining information about what methods are executed from trace scenarios with information obtained from the comments and identifiers in the source code. Latent Semantic Indexing is used to create indexes for methods that are executed by a particular scenario that implements a feature. A list of methods is ordered by similarity based on the method and the feature description (i.e. user query) is produced. The user selects if a method is related to the feature or not. Two case studies have been used to evaluate the approach in terms of performance and usability in comparison when only latent semantic indexing method is used to identify traceability relations between parts of code and a feature. The results demonstrate that the hybrid approach is more effective in terms to show relevant methods in top position than when using only latent semantic indexing method.

Egyed et al. (Egyed, 2003), (Egyed, et al., 2005) proposed a scenario driven approach for traceability generation. The approach requires an observable and executable software system, scenarios describing the use of the system, and a set of initial hypothesized trace relations between software artefacts and the scenario. The traceability relations are generated and validated by executing scenarios on the running software system and observing the lines of the code that are used. The parts of the code that are executed by the scenarios (*footprints*) are represented as a graph structure (*footprint* graph) showing the overlaps between the scenarios and the parts of the code. The *footprint* graph is normalised and refined. The traceability relations are created based on transitivity dependency relations (e.g., if A depends on B, and B depends on C, then A depends on C) and usability dependency relations (e.g., if A uses a sub set of part of the code that B uses then A depends on B).

Run-time information is also used by Grechanik's approach (Grechanik, et al., 2007). The approach addresses the problem that some programmers give arbitrary names to elements in the code. Grechanik et al. combine information obtained from runtime values of entities in the program with static information from the source code and use case diagrams. As in (Egyed, 2003), (Egyed, et al., 2005), a set of initial traceability relations between use case diagram elements and source code is required. The experiments show that in average it is sufficient to identify manually less than seven percent of traceability relations of a program to give an acceptable accuracy.

2.2.7 Hypermedia and Information Integration approaches

Hypermedia and information integration approaches rely on the integration of tools to create traceability relations. The main drawback of this approach is that can be difficult to make the integration in an environment where there is a great variety of tools been used and the tools used to develop software can change constantly.

TraceM (Sherba, et al., 2003), (Spanoudakis, et al., 2005) is an approach that makes use of services provided by open hypermedia technology to the creation of traceability relations.

TraceM coordinates the use of the Infinite open media system and the Chimera information integration system. The process involves extracting the relations from the Infinite tool and then import into the Chimera system.

2.3 Representation, Recording and Maintenance of Traceability Relations

Several different approaches have been used to represent, record and maintain traceability relations such as database, hypermedia, ontology, and mark-up approaches (Spanoudakis, et al., 2003) (Sherba, 2005), (Gotel, et al., 1994), (Maletic, et al., 2003), (Spanoudakis, et al., 2005), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009) and (Sharif, et al., 2007).

The database approach is used by most of the requirement management tools such as IBM DOORS (IBM Rational, 2009), IBM Rational RequisitePro (IBM Rational, 2009), and Borland Caliber (Borland, 2009). The database approach has the advantage of using all the facilities provided by database systems and database management systems (e.g., query facilities, consistency control, fault tolerance, and concurrency control), which are important in industrial settings.

TraceM (Sherba, et al., 2003), (Sherba, 2005) uses a hypermedia approach and supports the creation of n-ary relations between different types of objects. The approach uses metadata information about the documents being compared. Open hypermedia relations are stored separated from the associated elements and hypermedia systems are used to manage the relations. Sherba et al. argue that data model of hypermedia systems is more powerful than the data model provided by mark-up language such html that consists of one-way pointers embedded inside documents while data model of hypermedia system allow n-ary links between multiple sets of documents.

Mark up approaches has been proposed by Gotel (Gotel, et al., 1994) and Maletic (Maletic, et al., 2003). The main advantage of these approaches is that relations can be visualised by Internet browsers. The approaches described in (Spanoudakis, et al., 2005), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009) and (Sharif, et al., 2007) use XML to represent traceability relations. In each of these approaches, the XML elements and attributes used to represent a traceability relation are different. The traceability relations in (Alves-Foss, et al., 2002) are represented using XML Linking Language (XLink).

Burge et al. (Burge, et al., 2007) implements Software Engineering Using RATionale (SEURAT) tool that supports the representation of rationales. Burge technique uses an Argument Ontology approach to represent the rationale instead of a goal oriented technique. SEURAT tool capture the relations between functional and non-functional requirements using an ontology representation. Traceability between lower level and higher level goals is provided indirectly by the hierarchy of non-functional requirements in the ontology. The ontology defines relations between different goal levels and rationale.

2.4 Visualisation of Traceability Relations

The simplest ways to show traceability relations are using traceability matrix, tree browser explorer, and hyperlinks views (Marcus, et al., 2005). Some of the tools also provide query and filter facilities to enable more specific searches (Borland, 2009), (IBM Rational, 2009), (IBM Rational, 2009). The traceability matrix is used when an overall view of traceability relations is required. Tree browser helps to navigate through specific relations, and hypermedia views are used when the objects and relations need to be presented in the same view.

The current way in which commercial tools support visualisation, analysis and management of traceability information is limited. Some approaches have been proposed in order to improve this situation. Hollings et al. propose in (Hollings, et al., 2005) to use *XML Topic Maps (XTM)* (TopicMaps.org, 2001): an abstract model and a XML grammar for interchanging Web-based topic maps to represent requirements traceability. The use of XML Topic Maps improves the management, analysis and visualisation of requirements. Sherba et al. propose in (Sherba, et al., 2003), (Sherba, 2005) an approach where traceability relations can be visualised in the same tool where the artefacts are originally created, using open hypermedia and information integration techniques. Marcus et al. discuss and propose in (Marcus, et al., 2003) a set of high level requirements for a tool that support visualisation of traceability relations. They also present a TraceViz tool that implements some of these requirements: i) integration with an IDE; ii) interface or integration with traceability relation recovery tools; iii) visualisation and recording of traceability relation among various artefacts independently of the data representation of the artefacts; iv) provide flexible and user customizable views of the traceability data;

TraceViz is integrated into the Eclipse IDE as a plug-in and there is an easy navigation between the TraceViz view and the source code. TraceViz is integrated with a tool presented in (Marcus, et al., 2005), which uses latent semantic indexing to recover traceability relations between source code and external documentation. Traceability relations in TraceViz are stored in a simple XML format. TraceViz support different views such consistency-based view that groups together relations whose elements change or did not change and artefact-based view that groups together relations with the target of the same type and the user can also define additional views and categories, based on other attributes, or groups of attributes.

2.5 Use of Traceability Relations

Traceability relations have been used to support the development of software systems in several activities such as change management, impact analysis, system verification, validation, testing, and reuse of software artefacts (Mohan, et al., 2008), (Sharif, et al., 2007), (Murta, et al., 2006), (Murta, et al., 2006), (Cleland-Huang, et al., 2003), (Grechanik, et al., 2007), (Kurtev, et al., 2007), (Goknil, et al., 2008), (ten Hove, et al., 2009), (Asuncion, 2007), (van den Berg, et al., 2006).

Mohan et al. (Mohan, et al., 2008) highlight the importance to integrate software configuration and traceability practices in order to support software evolution.

Sharif et al. (Sharif, et al., 2007) present an approach to support software evolution using traceability relations. Models are represented in XML and a tool is used to identify differences between versions of models. The output of the tool is an XML file that contains common and different parts of the different versions. Suspected relations are identified by the tool using the information about what has changed. The relation can be updated manually by the user or automatically by some tool depending of the case.

ArchTrace is a tool that supports software evolution (Murta, et al., 2006a), (Murta, et al., 2006b). Some policies rules can be created to respond to changes in the software system. In particular, ArchTrace supports software evolution between architecture description using xADL and source code stored in Subversion configuration management system. The approach requires an initial definition of a set of relations.

Huang et al. also describe an approach to support software evolution (Cleland-Huang, et al., 2003). The approach is based on Event Notifier design pattern. Initially, new requirements are registered in a server and artefacts related to a requirement can subscribe to receive notifications when some modifications occur to that requirement. A modification on the requirement triggers an event that is handled by a server. The server processes the event and sends a notification message to all dependent artefacts. The type of message depends on the artefact and the semantic of the relation.

Grechanik et al. implement a tool that allows the user to navigate between traceability relations created by their approach (Grechanik, et al., 2007). The user can navigate from use cases to the related parts of the source code and vice-versa.

Kurtev discusses the use of traceability relations to support change management and impact analysis in Model Driven Engineering development (Kurtev, et al., 2007). Two types of approaches can be used accordingly to Kurtev when changes occur. One approach is to re-execute the transformation from the whole source model to the target model and another approach is to execute incremental transformation by performing transformations of only those elements that have changed in the source model. In (Goknil, et al., 2008), Goknil uses the semantic of traceability relations to support change and impact analysis of requirements. Several rules are created to be executed when requirements are created, deleted or modified depending of the relations between the artefacts (*requires*, *refines*, *conflicts*, and *contains*). ten Hove et al. shows (ten Hove, et al., 2009) a tool that supports impact analysis based on traceability relations defined by SysML (*composedBy*, *deriveReqt*, and *copy*) between requirement from the domain and from the model. Some rules are created to propagate changes and to identify inconsistencies between the model and the domain.

Assuncion et al. present an experiment (Asuncion, et al., 2007) in an industrial case study that demonstrates that traceability can be used to enforce process adherence and as a consequence to comply with standards required by customers.

Traceability relations have been used to identify crosscutting relations dependencies between software artefacts (van den Berg, et al., 2006). van den Berg et al. describe an approach where crosscutting relations through the whole software development process can be identified based on traceability relations defined between source and target models and based on transitivity properties of those relations. For instance, van den Berg shows that crosscutting relations between concerns and design elements can be identified based on traceability relations between concerns and requirements and based on traceability relations between requirements and design.

2.6 Traceability Approaches for Multi-Agent Systems

The majority of the work in the area of agent-oriented software engineering has been concerned with the development of methodologies, notations, and programming languages (Luck, et al., 2004).

In (Bordini, et al., 2005) and (Luck, et al., 2004), the authors discuss the current state-of-the-art and issues of agent oriented software development and suggest directions for future research in this area. As stated in these references, very little has been done to support traceability in agent oriented software engineering.

In (Pinto, et al., 2005), the authors propose different types of traceability relations and a traceability reference model, and define an agent oriented software process that extends Tropos methodology to support traceability generation.

In (Perini, et al., 2005), traceability relations are generated automatically during the transformation process between source and target model elements for Model-Driven Architecture using Tropos methodology.

In (Padgham, et al., 2004), Padgham et al. define how to check consistency and completeness between elements in Prometheus design models. Some of these checks have been implemented in PDT tool (Padgham, et al., 2005). The PDT tool has also been extended in (Padgham, et al., 2007) to support automatic generation of skeleton code in JACK. The work in (Padgham, et al., 2005) supports consistency and completeness checks, but limited to design phase.

As described above, existing approaches to support traceability relations in multi-agent systems are still in their initial stages and limited. To the best of our knowledge, existing approaches to assist with automatic generation of traceability relations do not provide support for different types software models created during different phases of the development life-cycle of multi-agent systems such as requirements, design and implementation phases. Moreover, they do not offer support to assist with the identification of missing elements and discrepancies of names among the artefacts created. Furthermore, existing approaches do not provide traceability relations of different types with explicit semantic meanings and do provide ways to communicate confidence level of traceability relation.

2.7 Performance Measures

The set of retrieved relations does not in general coincide with the set correct relations. It happens that the traceability method will fail to retrieve some of the correct relations, while on the hand it also retrieve traceability relations that are not relevant. Several measures have proposed to measure performance of traceability approaches. Recall and precision are widely used to demonstrate effectiveness of traceability approaches (Marcus, et al., 2003), (Marcus, et al., 2005), (Spanoudakis, et al., 2004), (De Lucia, et al., 2004). Precision can be seen as measure of exacteness, where recall is a measure of completeness.

Marcus et al (Marcus, et al., 2003) discuss the importance of precision and recall.They say that there are times when the user needs recovery all the correct relations even that means recovering many incorrect ones at the same time. Other times, precision is preferred and the user restricts the search space so all the recovery relations will be correct.

During the process the recovery of traceability relations, the user has the possibility to determine a threshold ϵ for the similarity measure, which identifies which elements are considered related. Only the pairs of elements that have a similarity measure greater than ϵ is considered a traceability relation. Choosing a higher threshold will result in higher precision while lowering the threshold will increase recall. Therefore, evaluation can be made based on the precision/recall curves on retrieval tasks varying the threshold

A second option for the user is to impose a threshold on the number of recovery links, regardless of the actual value of the similarity measure. The user can choose to select the top n relations for each pair of elements.

Hayes et al. (Hayes et al., 2007) evaluate a traceability approach using students divided in two groups: those doing manually traceability and those doing traceability using the traceability tool (i.e. RETRO). It was record the amount the time spent on the task by each group and additionally asked the students to answer a survey with questions specific to the nature of the process employed by each group.

The results showed that took the manual group almost three times as long to complete the task (120.66 min as compared to 41.8 min.) as the group that had used the tool. Students that used the tool found a higher percentage of the correct links than the students that realised the tasks manually (70.1% recall versus 33% recall). The students doing manual tracing built RTMs with much higher precision (24,2% as compared to 12.8%) than the group using the tool.

The students that used the tool answered a survey about what featured they had used and how happy they were about it. The students gave an overall positive impression of the features that they used and agreed that the tool was reasonably easy to use.

De Lucia et al. (De Lucia, et al., 2007) also used 20 master students to respond the questions if traceability tools improve the tracing performance of the software engineer with respect to manual tracing and if they reduce the time spent by the software engineer to trace links. The results achieved in the experiments demonstrate that the use of traceability recovery tool significantly improves the tracing performance of software engineer.

2.8 Implication of tools that infer trace relations

The task of maintaining relations among software artefacts is tedious and time consuming. Tools that infer trace relations reduce the time spent in the task to create RTMs matrix. Although there are several commercial tools available that support traceability between artefacts (IBM Rational, 2009a) (IBM Rational, 2009b)(Borland, 2009). The main drawback of these tools is the lack of automatic traceability generation. The need for tools that support traceability recovery has been recognised in the last years (Spanoudakis, et al., 2005).

Assuncion et al. (Assuncion et al., 2007) discuss benefits of successful end-to-end software traceability tool developed by a software development company. Traceabily aids in system comprehension, impact analysis, system debugging and communication between the development team and stakeholders.

Assuncion et al affirms that relations between artefacts can be interwined with the underlying software processes. Raising the visibility of actual software enables users to compare actual practices to stated company procedures. Therefore, traceability offers the potential for improving the actual software process, as well as capturing the rationale behind a specific artefact, fostering system comprehension.

The advantages to use tool to capture traceability relations is that is faster to identify traceability relations using tool than to identify traceability relations manually. Even when the tools identify traceability relations incorrect, it is easier to point out what are the incorrect traceability relations than identify all traceability relations manually.

2.9 Summary

This chapter describes what is traceability and its importance. It discusses about different types of classifications for traceability relations that have been proposed in the literature. It presents different approaches to capture, store and use traceability information. It shows how traceability relations can be visualised and queried by different approaches. Finally, the chapter discusses traceability approaches that have been applied to the area of multi-agent systems.

Bordini et al. (Bordini et al., 2007) discusses what are the key current issues in developing multi-agent systems and what should the research community should concentrate future research efforts. They identify three key areas for future work. These areas were techniques for integrating design and code; extending agent-oriented programming languages to cover certain aspects that are currently weak or missing (e.g. social concepts, and modelling the environment); and development of debugging and verification techniques.

As discussed in this chapter, software traceability is essential in the software development process and helps in several activities such as impact analysis, software maintenance and evolution, component reuse, verification, and validation. In the following two chapters, we describe a traceability reference model for multi-agent systems developed using *i** framework, Prometheus methodology and JACK language and a traceability framework to identity missing information and automatically generate traceability relations between elements created during the development of multi-agent systems.

Chapter 3 - Traceability Reference Model

In this chapter, we present a traceability reference model for elements created during the development of multi-agent systems using *i** framework (Yu, 1995), Prometheus methodology (Padgham, et al., 2004), and JACK language (Agent Oriented Software Limited, 2010).

We describe *i** framework, Prometheus methodology, and JACK language elements used by our framework in detail. We explain why *i** framework, Prometheus methodology, and JACK language are important when developing multi-agent systems, and describe traceability relations between elements in *i** and Prometheus and between Prometheus and JACK. Throughout the chapter, we present examples to illustrate our traceability reference model.

3.1 Overview of the Reference Model

A traceability reference model is concerned with different software models created during the development of software systems, their respective artefacts, and the relations that exist between these various artefacts. The use and importance of traceability reference models have been discussed in Section 2.1 (Chapter 2). As we mentioned in Chapter 2, the need to capture the different types of traceability relations and their semantics has been pointed out as fundamental by several authors (Aizenbud-Reshef, et al., 2006), (Dick, 2002), (Ramesh, et al., 2001), (Spanoudakis, et al., 2005) in order to make effective use of traceability and to support deeper types of software analysis (e.g. change and impact analysis activities).

Several methodologies have been proposed to support the development of multi-agent systems such as Prometheus (Padgham, et al., 2004), Tropos (Castro, et al., 2002), and MaSE (DeLoach, 2001), and Gaia (Wooldridge, et al., 2000). In addition, goal modelling, specification and reasoning have been recognised as fundamental activity of the software development process (van Lamsweerde, 2001). Goals are used in early phase of the requirement engineering process. In Agent Oriented Software Engineering, goal is a basic concept and it is used in the implementation of several agent architectures (e.g. BDI architectures (Rao, et al., 1992)). Business modelling techniques (Yu, 1995) have also been proposed to specify models of goals, structures, and processes of organizations. Such models can be applied to assist stakeholders and developers to develop a common understanding of the organization and, therefore, facilitate the identification of requirements for systems that are developed to support its function.

Our traceability reference model concentrates on models and artefacts generated when using the *i** framework (Yu, 1995), Prometheus methodology (Padgham, et al., 2004), and JACK language (Agent Oriented Software Limited, 2010), (Busetta, et al., 1999), (Howden, et al., 2001).

We adopt the Prometheus methodology (Padgham, et al., 2004) in our work due to several reasons, namely (a) Prometheus is largely used in both industrial and academic settings; (b) Prometheus provides cover to the whole development life cycle of multi-agent systems; (c) Prometheus offers a detailed design phase with concepts that match concepts in JACK; and (d) there is good support on using Prometheus in terms of documentation, tools, and examples.

We use *i** framework (Yu, 1995) in order to enhance, Prometheus methodology by allowing support for the early requirements phase for both goals and business modelling. *i** framework has been recognized as one of the most important requirement engineering techniques to represent business model, and it is a simple technique which contains all necessary elements to represent organizational models.

Our work adopts JACK (Agent Oriented Software Limited, 2010) due to its use in several academic and commercial applications (e.g., unmanned aerial vehicles, surveillance, air traffic management, real-time scheduling, and virtual actors) (Agent Oriented Software Limited, 2010), and large documentation support. Moreover, the design phase of Prometheus describes in detail the elements of the JACK.

We propose nine types of traceability relations between the artefacts in *i**, Prometheus, and JACK code. The different types of traceability relations that we propose is based (i) on our experience of using *i** framework, Prometheus methodology, and JACK language to develop multi-agent systems; (ii) on the semantics of the artefacts in *i** framework, Prometheus methodology and JACK language; (iii) on our study and experience with software traceability, and (iv) on the various types of traceability relations proposed in the literature (Davis, 1990), (Gotel, et al., 1994), (Lindvall, et al., 1996), (Dick, 2002), (Ramesh, et al., 2001), (Spanoudakis, et al., 2005), (Almeida, et al., 2007), (Toranzo, et al., 1999), (Han, 2001), (Pinto, et al., 2005), (Pohl, 1996), (Jirapanthong, et al., 2009), (Rilling, et al., 2007). The different types of traceability relations that we propose are: overlaps, contributes to, uses, creates, achieves, depends on, composed of, sends and receives.

In the next sections we describe details of the *i** framework, Prometheus methodology, JACK language, and different traceability relations.

3.2 Multi-agent Oriented Artefacts

3.2.1 i* Framework

The *i** framework was developed by Yu (Yu, 1995) to provide a richer modelling technique to represent organizational process. The main concept of the *i** framework is an actor. The framework is composed of two types of diagrams called (a) the Strategic Dependency Model (SD) and (b) the Strategic Rationale Model (SR). The Strategic Dependency (SD) and Strategic Rationale (SR) models are used to represent the intentional relations among organizational actors. In this sub-section we describe some main concepts of *i** framework. A detailed description of *i** can be found in (Yu, 1995).

The SD model consists of a network of dependencies between actors. It is composed of five types of elements namely *actors*, *goals*, *soft goals*, *resources*, and *tasks*, and four types of dependencies, namely *goal dependency*, *resource dependency*, *task dependency*, and *soft goal dependency* (see Figure 3.1) (Yu, 1995).

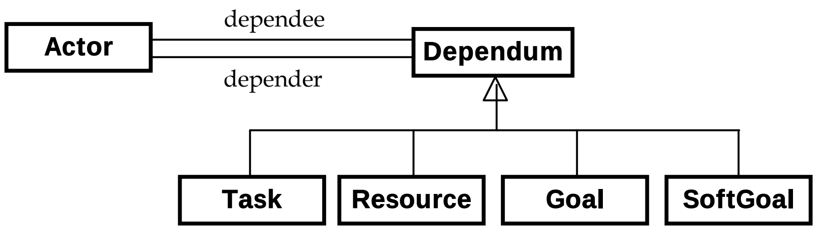


Figure 3.1 SD model

In the goal dependency, an actor (*depender*) depends on another actor (*dependee*) to bring about a certain condition or state in the world, while the *dependee* is free to choose how to deliver the goal. The condition concerned with the achievement of a goal is precisely defined and its satisfaction can be clearly evaluated to true or false. In the task dependency, the *depender* depends on the *dependee* to carry out an activity and differently of a goal dependency, in the task dependency, how the *dependee* executes a task is important to the *depender*. The softgoal dependency is similar to a goal dependency. It represents a condition in the world that an actor would like to achieve. However, differently from the goal dependency, the criterion for the condition to be achieved is not clearly defined. For certain goals it is difficult to decide a clear-cut definition if the goal have been achieved or not, because the meaning of the goal is not clear or its satisfaction is defined subjectively. A resource dependency means that one actor depends on another actor for the availability of an entity (physical or informational).

Figure 3.2 shows a Strategic Dependency diagram for part of the Electronic Bookstore case study used to evaluate our work (see Chapter5). In the Figure, actors are represented by circles, goals are represented by ovals, and resources are represented by rectangles. Figure 3.2 shows that a Customer actor depends on the Electronic Bookstore actor to achieve its goals of being supplied with a wide variety of books (Supply wide variety of books), browsing books (Browse Book), buying books (Buy Book), searching books (Search Book), and tracking the delivery of book orders (Track Delivery). The Delivery Manager and Sales Manager actors depend on the Electronic Bookstore actor to monitor the shipping of books (Monitor Shipment) and to keep prices at a competitive level (Keep Prices Competitive), respectively. The Electronic Bookstore actor depends on the Delivery System actor to deliver books (Delivery Shipment), to be informed of lost or damaged items (Inform Lost or Damaged Item), to track shipment (Track Shipment), and to be informed when a book has or has not been delivered (Inform Book Delivery and Inform Delivery Refused, respectively). The Electronic Bookstore actor also depends on the Customer and Bank actors to receive the information about a Delivery Choice, Credit Card Details, and Bank Transaction Response (represented by the respective resources). The Customer actor depends on the Electronic Bookstore to have the confirmation of a book order (Book Order). The Stock Manager depends on the Electronic Bookstore to control the stock (Control Stock) and to update the catalogue of books (Update Catalogue of Books). The directions of the dependencies are represented by the arrows.

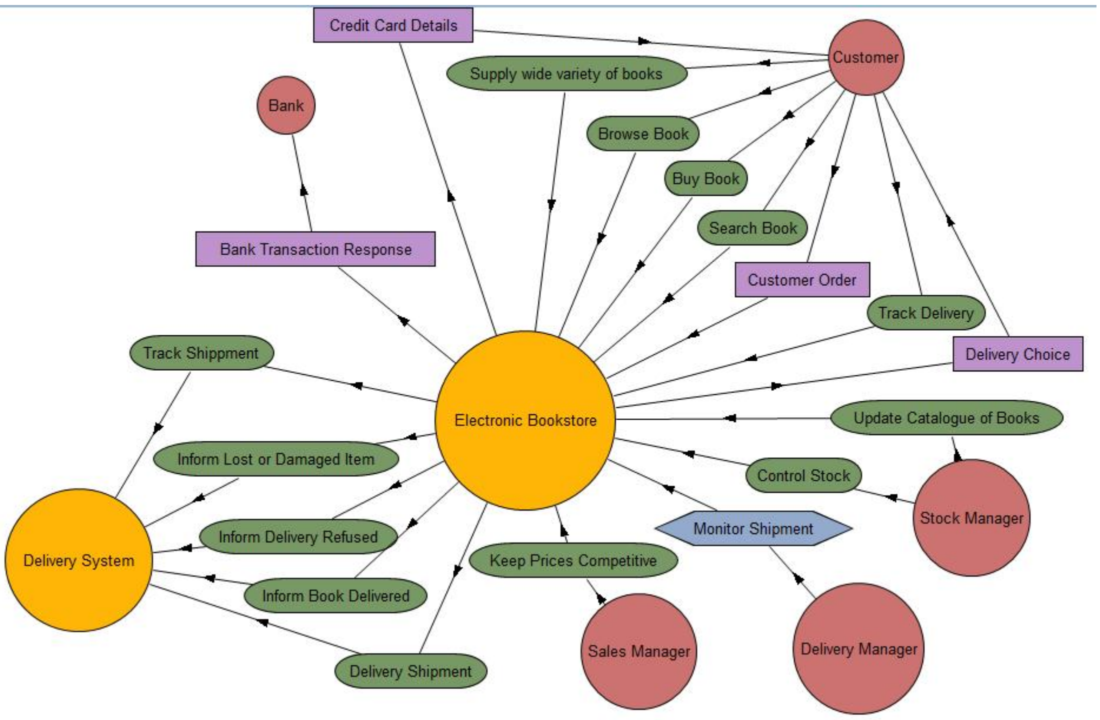


Figure 3.2 Strategic Dependency Diagram for the Electronic Bookstore

SR model in *i** describes the process of how an actor achieves its associated dependencies represented in the SD model, and the rationale (*why*) behind these dependencies. The SR model provides four elements namely goal, task, resource, and soft goals, and two types of relations called *means-end* and *task decomposition*. A means-end relation denotes an association between an element representing the end (i.e. goal, task, resource, or soft goal) and means of achieving this element. A task decomposition relation describes how a task can be executed in terms of sub-components, such as resources, goals, soft goals, and tasks.

Figure 3.3 shows a partial SR diagram for the Electronic Bookstore actor. In Figure 3.3, tasks are represented by angled rectangles, goals are represented by ovals, and resources are represented by rectangles. Figure 3.3 describes how the Electronic Bookstore achieves Browse Book, Buy Book, Track Delivery, Monitor Shipment, Control Stock, Search Book, and Keep Prices Competitive, and Supply wide variety of books dependencies. The Electronic Bookstore actor provides Browse by New Release, Browse By Category, Browse By BestSeller and Browse By Special Offer as means to achieve Browse Book. A Customer can place an order on-line (Place Order Online sub-task) or place an order by phone (Place Order By Phone) to buy a book. The Place Order Online task is decomposed on Update Customer Orders, Delivery Handling, Make Payment and Send Book Order Confirmation sub-tasks. The Electronic Bookstore actor needs Customer Order resource to Update Customer Orders. The Delivery Handling task is decomposed on Fill Pending Order and Organize Delivery. The Organize Delivery task is decomposed on Place Delivery Request, Compute Delivery Time Estimates, and Obtain Delivery Options and Log Outgoing Delivery. The Electronic Bookstore actor uses Postal DB and Courier DB to Obtain Delivery Options. The Electronic Bookstore uses Customer DB to Compute Delivery Time Estimates. The Make Payment task is decomposed on Obtain Credit Card Details and Perform Bank Transaction. A transaction can be accepted (Transaction Accepted task) or rejected (Transaction Rejected task). The Electronic Bookstore achieves Track Delivery goal by using the Determine Delivery Status and Location sub-task. The Electronic Bookstore actor provides Determine Delivery Status and Location, Log Delivery Problems and Update Delivery Status as means to achieve Monitor Shipment goal. The Electronic Bookstore actor provides Log Outgoing Delivery, Log Books Arriving, and Restore Stock as means to achieve Control Stock goal. The Electronic Bookstore needs to Set Prices Competitively, Re-establish Book Price, keep Lower Book Price and Monitor Bookstore to achieve its goal to Keep Prices Competitive. As means to Supply wide variety of books the electronic bookstore needs to Update Catalogue of Books. A book can be searched by name (Search By Name) or by more specific criteria (Advanced Search). The Electronic Bookstore provides Search By Author, Search By Title, Search By Subject and Search By ISBN as means to achieve the Advanced Search goal.

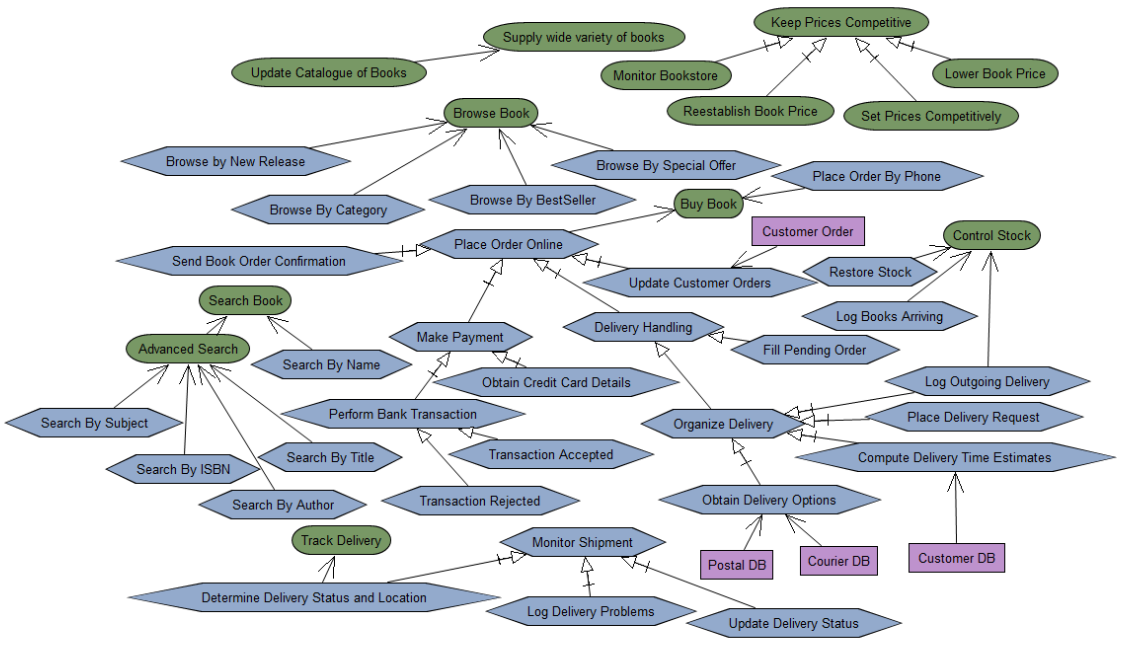


Figure 3.3 Strategic Rationale Diagram for the Electronic Bookstore actor

3.2.2 Prometheus

Prometheus (Padgham, et al., 2004) is a methodology developed by the RMIT University in collaboration with the Agent Oriented Software Ltd. Prometheus methodology provides several diagrams and descriptors to describe analysis and design phases of multi-agent systems such as Goal diagram, Role diagram, Use Case Scenario, System Overview diagram, Agent Overview diagram, Capability diagram, Process diagram, and Protocol diagram. In this section we describe some of the main concepts of the Prometheus methodology. A detailed description of Prometheus can be found in (Padgham, et al., 2004).

The methodology consists of three phases, namely: system specification, architectural design, and detailed design phases, as shown in Figure 3.4 (Padgham, et al., 2004).

The system specification focuses on (a) identifying the system goals, the basic functionality of the system, the interface between the system and its environment in terms of inputs (percepts) and outputs (actions); and (b) developing use case scenarios that are similar to scenarios used in object-oriented approaches with a slightly enhanced structure.

The architectural design phase focuses on (a) deciding what agent types the system will contain; (b) developing the agent descriptors; (c) capturing the structure of the system by using a system overview diagram that models relations between agents, events, and shared data objects; and (d) describing the dynamic behaviour of the system. The behaviour of the system is described using interaction diagrams and interaction protocols.

The detailed design phase is concerned with describing the agents in terms of capabilities, events, plans, and data structures. The artefacts generated in this phase are agent overview diagrams, capability overview diagrams, and descriptors for plan, data, and events.

The system specification phase shows the interactions between a multi-agent system and the environment where the system is being executed. This phase consists of identifying actions and percepts (i.e., outputs and inputs, respectively), and describing goals, roles and scenarios.

Goals are represented graphically using a goal diagram where top-level goals are refined by sub-goals. Figure 3.5 shows an example of a goal diagram for the Electronic Bookstore case study used to evaluate our work (see Chapter 5). As shown in the figure, the Order book goal is refined by the Place order (online) goal that is then refined by Make payment online, Update customers orders, Fill pending order and Arrange delivery.

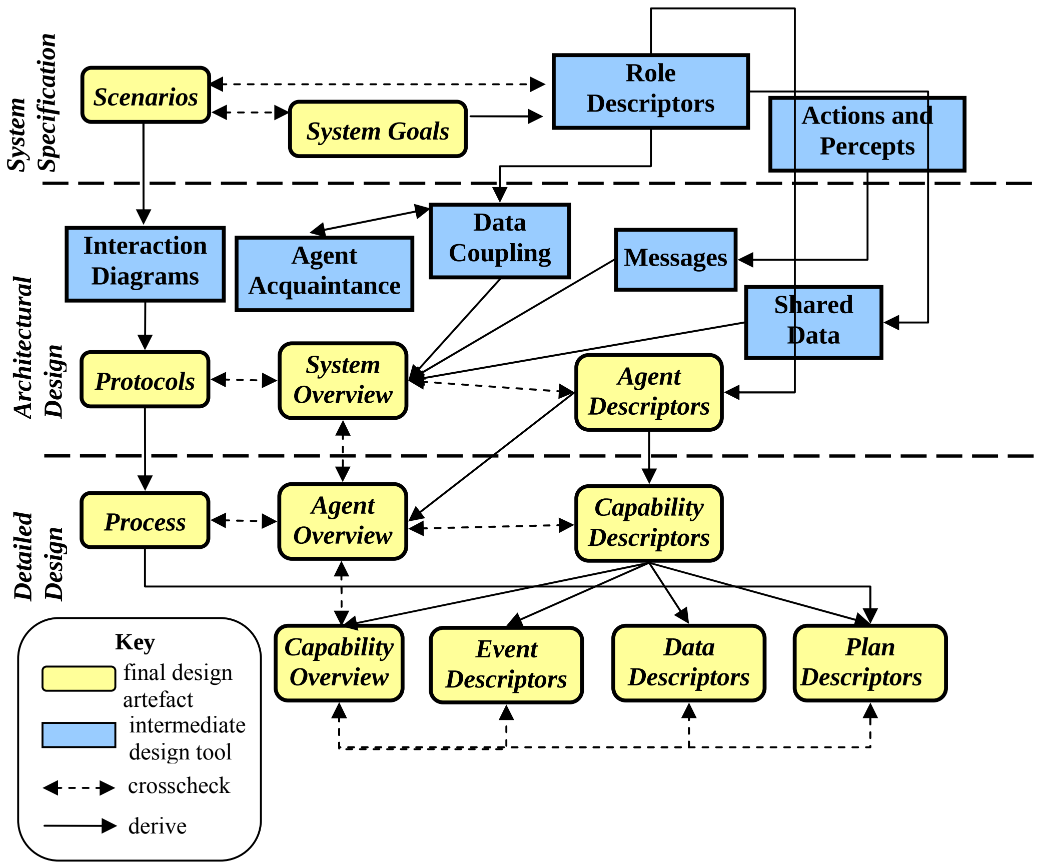


Figure 3.4 Prometheus methodology phases

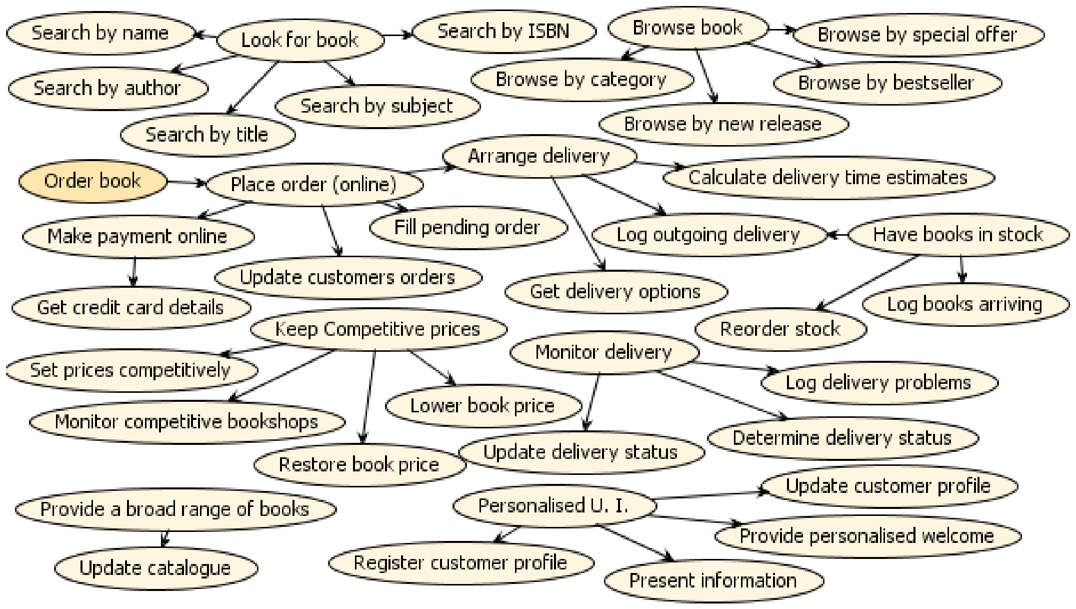


Figure 3.5 Goal diagram for the Electronic Bookstore

The concept of roles (or functionality) is used in Prometheus to group together goals, percepts, actions, and data related to some behaviour of the system. Figure 3.6 shows a role diagram for the Electronic Bookstore case study. The diagram shows Book Finding, Online Interaction, Competition Management, Customer Profile, Purchasing, Delivery Handling, and Delivery Management roles (represented as rectangles). Every role in the diagram is associated with percepts to which the role responds (represented as star elements), all actions that the role performs (represented as arrow rectangles), all information (data) used or produced by the role (not represented in the diagram), and all goals that the role achieves (represented as ovals). For instance, the Delivery Handling role that manages delivery of orders to customers (a) responds to the Delivery Choice percept, (b) performs the Place delivery request action, and (c) achieves the Arrange delivery, Calculate delivery time estimates, and Get delivery options goals.

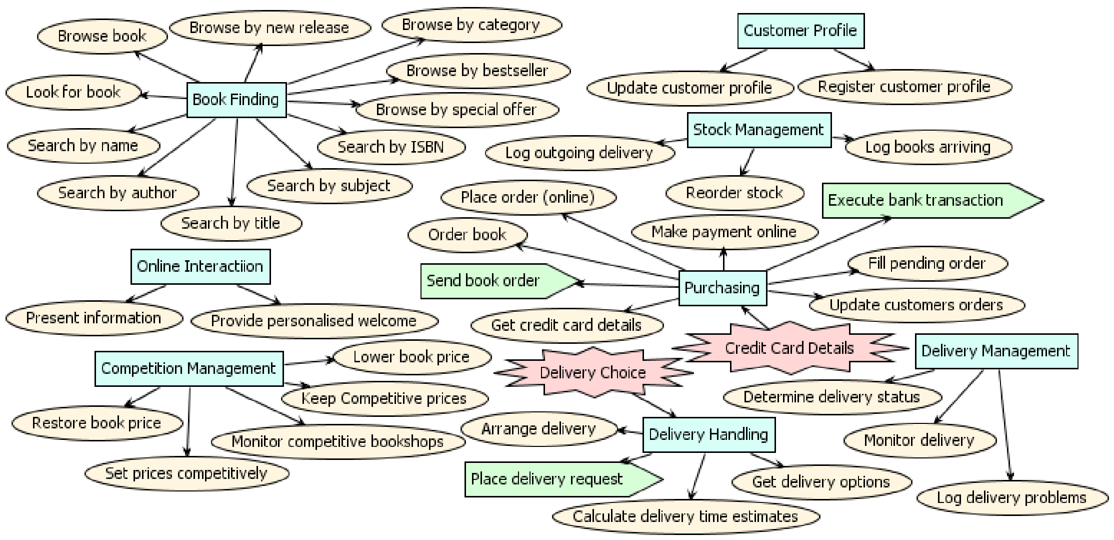
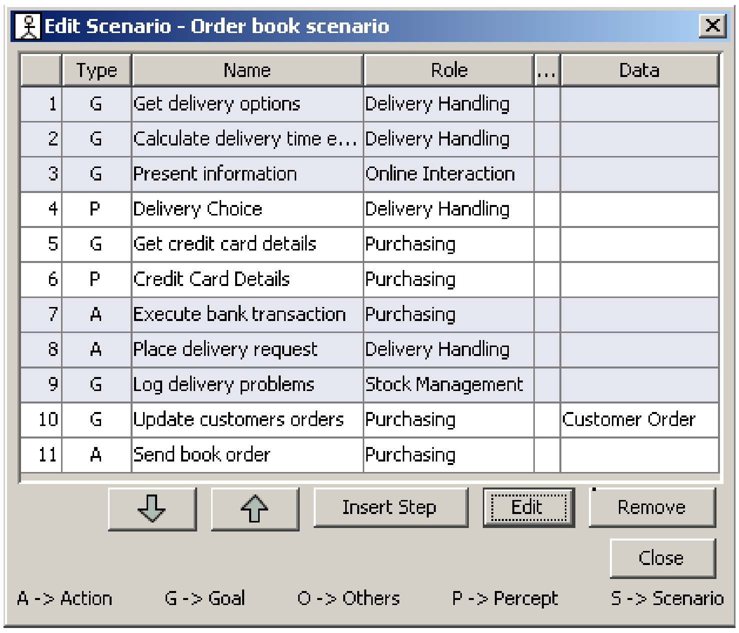


Figure 3.6 Role Diagram for the Electronic Bookstore

The interactions between actions, percepts and roles are represented using use cases scenarios. As in object-oriented approaches, Prometheus adopts use case scenarios to represent the sequence of steps executed by the system in order to achieve a goal (G), perform an action (A), and to receive a percept (P). Figure 3.7 shows an example of Order Book scenario. The Order Book scenario represents a sequence of steps executed by the system to achieve the Order Book goal. The multi-agent system retrieves all delivery options (Get delivery options goal), calculate delivery time estimate (Calculate delivery time estimate goal) and show all the options available to the user (Present information goal). The user selects the delivery option (Delivery

Choice percept) and then the system processes the information and wait for the credit card details of the user (Get credit card details goal). The user enters the credit card details (Credit Card Details percept), the system processes the input and then sends a message to the bank to execute the transaction (Execute bank transaction action), the place a delivery request (Place delivery request action). If a problem occurs to deliver the order then the information is added to the log (Log delivery problems goal). The system update customer orders information (Update customers orders goal) and then sends a message to the customer with the confirmation of the order (Send book order action).



	Type	Name	Role	...	Data
1	G	Get delivery options	Delivery Handling		
2	G	Calculate delivery time e...	Delivery Handling		
3	G	Present information	Online Interaction		
4	P	Delivery Choice	Delivery Handling		
5	G	Get credit card details	Purchasing		
6	P	Credit Card Details	Purchasing		
7	A	Execute bank transaction	Purchasing		
8	A	Place delivery request	Delivery Handling		
9	G	Log delivery problems	Stock Management		
10	G	Update customers orders	Purchasing		Customer Order
11	A	Send book order	Purchasing		

A -> Action G -> Goal O -> Others P -> Percept S -> Scenario

Figure 3.7 Order Book Scenario

The main deliverables of the architectural design phase are Agent Descriptors, System Overview diagram and Protocols diagram. Agent Descriptors defines agent types of the multi-agent system and System Overview diagram and Protocols diagram describe how agent interacts with each other and with their environment. Agent types are defined grouping together roles that share common functionalities and use the same data based on criteria of cohesion and coupling. Interactions protocols describe all possible interactions defined by interaction diagram created based on use case scenarios. Interaction diagram use similar notation to Interaction diagrams in UML (OMG, 2010). The difference is that the interactions are between agents instead of objects. Interaction diagram uses notation defined on Agent

UML (Huget, et al., 2003) that is similar to the notation used to create protocol diagrams in UML (OMG, 2010). The system overview diagram specifies how agent interacts with each other sending messages and with the environment using actions and percepts. Figure 3.8 shows System Overview Diagram for the Electronic Bookstore. The main elements in this diagram are agents (represented as rectangles with an image of an actor inside the diagram), percepts (represented as star elements) to which the agents respond, actions performed by the agents (represented as arrow rectangles), messages exchanged between agents (represented as a rectangle envelope), and external data accessed by the agents (not shown in Figure 3.8).

As shown in Figure 3.8, the Electronic Bookstore multi-agent system is composed of agents Sales Assistant, Stock Manager, Credit Card Agent, and Security Manager. The Sales Assistant agent in Figure 3.8 responds to Book Details percept, which contains information about purchase of books, and to Keyword Search percept, which contains information about a search in the book catalogue. The Sales Assistant agent also sends Book Query and Book Purchase messages to Stock Manager agent requesting information about an item in the book catalogue and requesting a purchased item to be removed from the stock, respectively. The other elements in the diagram are represented similarly.

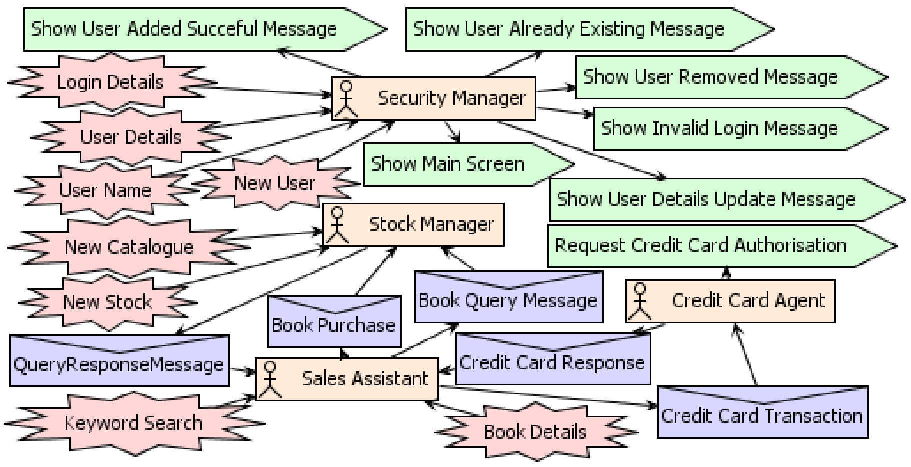


Figure 3.8 System Overview Diagram

The interactions defined on use case scenarios, interaction diagram, and protocol diagram are refined on the detailed design phase using a slight variant of UML (Rumbaugh, et al., 1999) activity diagrams to create process diagrams. In the detailed design phase, the internal structure of each agent is described in terms of capabilities, messages, data and plans. Capabilities can be used to group together actions, data, messages and percepts related. Figure 3.9 shows an example of Find BestSellers Capability. The capability contains BestSellers Request message, BestSellers Response message, Find BestSellers plan and bestsellers data. The Find BestSellers plan is triggered by the BestSellers Request message. The Find BestSellers plan processes BestSellers Request message, use bestsellers data and sends a message with the result of the bestsellers books (BestSellers Response message).

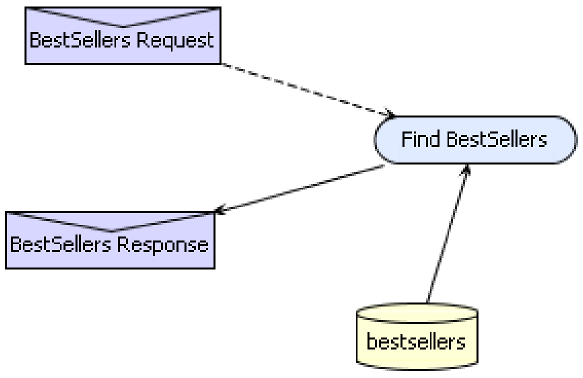


Figure 3.9 Find BestSellers Capability

The agent overview diagram represents in detail the design of an agent. The main elements in this diagram are actions (represented as arrow rectangles), plans (represented as ovals), data (represented as a data storage symbol), and percepts (represented as arrow rectangles). Figure 3.10 shows an example of an agent overview diagram for Security Manager agent. As shown in the Figure, in the presence of Login Details percept, the Validate User plan is executed. This plan consists of checking if a user's login information matches a data in the User DB data storage. In positive case, action Show Main Screen is executed; otherwise action Show Invalid Login Message is performed. The other elements in the diagram are represented similarly. A detailed description of Prometheus methodology can be found in (Padgham, et al., 2004). The Prometheus methodology is supported by the Prometheus Design Tool (PDT, 2010).

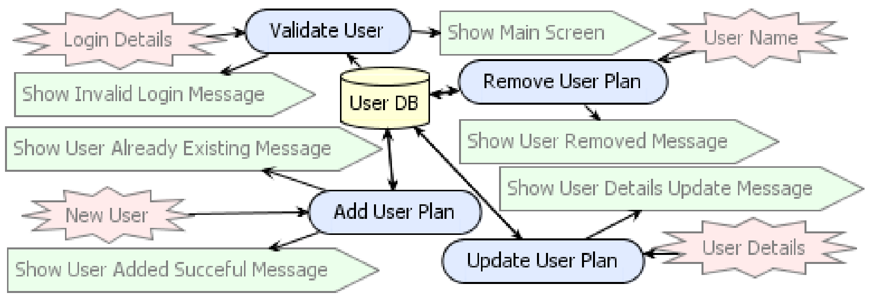

Login Details
Validate User
Show Main Screen
User Name
Show Invalid Login Message
User DB
Remove User Plan
Show User Already Existing Message
Show User Removed Message
Show User Details Update Message
New User
Add User Plan
User Details
Show User Added Succeful Message
Update User Plan

Figure 3.10 Security Manager Agent Overview Diagram

3.2.3 JACK

JACK language is part of the JACK Intelligent Agent Environment developed by the Agent Oriented Software Limited (Agent Oriented Software Limited, 2010). This section describes the JACK constructs used in our approach. More details about JACK language can be found in (Agent Oriented Software Limited, 2010), (Howden, et al., 2001), (Busetta, et al., 1999). JACK language extends Java language providing new constructs such as *agents, capabilities, beliefset, events* and *plans*.

An agent in JACK can use the same statements that are part of the Java language in addition to a set of declarations and methods provided by the JACK language. Declarations in JACK are followed by # symbol. Examples of declarations are: an agent can handle events, an agent can read or modify beliefs, and an agent can use plans, an agent post events and an agent send events to other agents. An example of a method is *achieve* that tells an agent to pursue a goal.

Figure 3.11 shows an example of the Airport agent that is part of the Air Traffic Control Environment case study that we have used in order to evaluate our work (see Chapter 5 for a description of the case study and Appendix C for complete JACK code of the case study).

```
import java.util.Hashtable;
import aos.jack.jak.event.TracedMessageEvent;

/** Airport agents.*/
agent Airport extends Agent {
    #has capability ArrivalSequencing seq;
    Airport(String name,String [] runway) {
      super(name);
      for (int i = 0; i<runway.length; i++)
          new Runway(runway[i],i);
      seq.enable(runway);
      TracedMessageEvent.tracer.start(this);
    }
}
```

Figure 3.11 Airport Agent in JACK

The first two lines in Figure 3.11 contain *import* Java statements followed by comments. An agent definition starts with the *agent* keyword followed by the type of the agent (e.g. Airport). All agents in JACK are sub-classes of the *Agent* class. The *Agent* class provides the core functionalities of an agent in JACK. In Figure 3.11 the Airport agent declares that it uses the *ArrivalSequencing* capability followed by the constructor.

Figure 3.12 shows an example of a *BankAgent* agent as part of the Automatic Teller Machine case study that we have used to evaluate our work (see Chapter 5 and Appendix B for more details). The first line in Figure 3.12 contains a *package* Java statement followed by an agent definition. The *BankAgent* agent can send *WithdrawResponse* events, handle *WithdrawRequest* events, execute *ProcessWithdraw* plans, read or modify *accounts* and *balances* beliefs.

```
package     agents;

public agent BankAgent extends Agent {

    #sends event WithdrawResponse response;

    #handles event WithdrawRequest;
    #uses plan ProcessWithdraw;

    #private data Accounts accounts("accounts.dat");
    #private data Balances balances("balances.dat");

    public BankAgent(String n)
    {
      super(n);
      try {
          if (accounts.nFacts() <= 0) {
            accounts.add(10, 10);
            balances.add(10, 1000);
          }
      } catch (Exception e) {}
    }
}
```

Figure 3.12 BankAgent agent in JACK

Capabilities are used in JACK to group together a set of beliefs that the capability can use, a set of events that the capability can post or send, a set of events that a capability can handle, and a set of plans that the capability can use. Capability is used to facilitate reuse and modularise code in JACK. Figure 3.13 shows an example of *ArrivalSequencing* capability. The first line contains an *import* Java statements followed by comments. A capability definition starts with the *capability* keyword followed by the type of the capability (e.g. ArrivalSequencing). All capabilities in JACK are sub-classes of the *Capability* class. The *Capability* class provides the core functionalities to capabilities. Figure 3.13 shows that the *ArrivalSequencing* capability handles *AircraftEvent* event, uses data from the *mutex* Java object and can execute *RequestSlot* plan. The *ArrivalSequencing* capability also contains the attribute *runways* and *getRunways* and *enables* methods.

```
import aos.jack.util.thread.Semaphore;

/**
   The ArrivalSequencing capability contains the handling of landing
   requests from aircraft through negotiation with available runways
   for an appropriate landing allocation.
*/
public capability ArrivalSequencing extends Capability {

    #handles external event AircraftEvent;

    #private data Semaphore mutex();

    #uses plan RequestSlot;

    String [] runways;

    String [] getRunways(){
      return runways;
    }

    void enable(String [] runways){
      this.runways = runways;
      mutex.signal();
    }
}
```

Figure 3.13 ArrivalSequencing Capability

Events are the way that agents react to percepts or respond to messages sent by other agents. JACK supports two types of events classified by *Normal* and *BDI* events. The process of selecting what plan to execute when a Normal event occurs is simpler than when a BDI event occurs. The process of selecting what plan to execute when a Normal event occurs involves finding the first plan that is *applicable* and *relevant* to handle that event. The process of selecting what plan to execute when a BDI event occurs involves a more elaborate reasoning and it can be customised to the solution of a specific problem. The failure in the course of execution of the plan also causes the reconsideration of alternative plans to achieve the goal. Normal events in JACK can be Event or Message Event. Normal Event is used when an agent post an event to itself or a changed belief triggers the event. Message Event is used when an agent wants to send a message to another agent. JACK provides several types of BDI events such as BDIFactEvent, BDIMessageEvent, BDIGoalEvent, InferenceGoalEvent, and PlanChoice. BDIFactEvent and BDIMesasgeEvent are similar to Normal Event and Message Event events, respectively. BDIGoalEvent represent goals that an agent wants to achieve. The

InferenceGoalEvent is used when it is desirable that all applicable plans to be executed. PlanChoice are used for meta-level reasoning to select what plan to be executed.

Figure 3.14 shows an example of the *WithdrawRequest* event. The first line contains a *package* Java statement followed by the import Java statement. The event definition starts with the *event* keyword followed by the type of the event (e.g. *WithdrawRequest*). The *WithdrawRequest* is sub-class of *MessageEvent* class. The *WithdrawRequest* event contains the attributes: *account*, *pin* and *amount*. The *WithdrawRequest* event contains the *withdraw* method that is called when the event is sent by an agent.

```
package agents;

import aos.jack.jak.core.Jak;

event WithdrawRequest extends MessageEvent {
    public int account;
    public int pin;
    public int amount;

    #posted as
    withdraw(int account, int pin, int amount)
    {
      Jak.log.log("WithdrawRequest:withdraw created");
      this.account = account;
      this.pin = pin;
      this.amount = amount;
      message = "withdraw["+account+","+pin+"]";
    }
}
```

Figure 3.14 WithdrawRequest event

Plans are the course of actions that are executed when an event occur. A plan in JACK is composed of two main parts. One part is used to identify what is the type of events that the plan responds, and if the plan is relevant and applicable. The other part defines the sequence of steps that are executed when the plan is selected.

Figure 3.15 shows the *WithdrawCash* plan. The first lines contain package and import Java statements. The event definition starts with the plan keyword followed by the type of the plan (i.e. WithdrawCash). Plans in JACK are sub-class of the Plan class. The #handles event declaration defines what type of event that the plan handles (i.e. Withdraw). The #uses agent declaration allows the plan to use methods defined by AtmClient interface. The agent that uses the *WithdrawCash* plan has to implement this interface. The context method is used during the process of selection to check if the plan is applicable.

```
package agents;

import gui.AtmInterface;
import gui.AtmClient;

public plan WithdrawCash extends Plan {
    #handles event Withdraw event;
    #uses agent implementing AtmClient atmc;

    context()
    {
      atmc.getHardware().cardInserted();
    }
    #sends event WithdrawRequest request;
    #reasoning method
    body()
    {
        @send( atmc.getBank(),
             request.withdraw( atmc.getAccount(),
                         atmc.getPin(),
                         atmc.getAmount())
             );
    }
}
```

Figure 3.15 WithdrawCash plan

The *#sends event* declaration determines what messages the plan can send to an agent (i.e. *WithdrawRequest*). The reasoning *body* method is called when the plan is selected to be executed.

Beliefsets are the way that an agent keeps the information about the environment where they are situated. JACK use relations to represent beliefSets and beliefs are stored in terms of tuples. A tuple is composed of zero or more fields that identify the tuple and zero or more fields that store values of the tuple. Relations can be stored as OpenWorld or CloseWorld. True and false beliefs can be stored in OpenWorld beliefsets and not found beliefs are interpreted as unknown information. Only true relations can be stored in ClosedWorld beliefsets and not found tuples are interpreted as false.

Figure 3.16 shows Accounts beliefset. The first line contains the package Java statement. The beliefset definition starts with the *beliefset* keyword followed by the type of the belief (i.e. *Accounts*). The Accounts beliefset is sub-class of the OpenWorld class. The *#key field* declares *account* as the identifier for the *Account* beliefset and *pin* is the value stored in the relation. The Accounts belieftset contains addfact, newfact, endfact, delfact, modfact, modbd that are called beliefset callbacks in JACK. They are invoked when some changes occurs in the beliefset. For instance, delfact is called when a tuple is removed from the beliefset.

```
package agents;

public beliefset Accounts extends OpenWorld {

   #key field int account;
   #value field int pin;
   #indexed query query(int i, int j);

   public void addfact(Tuple t, BeliefState d) {…}
   public void newfact(Tuple t, BeliefState d, BeliefState old){…}
   public void endfact(Tuple t, BeliefState old, BeliefState d){…}
   public void delfact(Tuple t, BeliefState d){…}
   public void modfact(Tuple t, BeliefState d, Tuple tr, Tuple fl){…}
   public void moddb(){…}
}
```

Figure 3.16 Accounts beliefSet

3.3 Traceability Relations

We have identified nine different types of traceability relations between the various elements in the models used in our approach. The types of traceability relations are overlaps, contributes to, uses, creates, achieves, depends on, composed of, sends and receives.

The classification is based on the types proposed in the literature (Spanoudakis, et al., 2005) such as overlaps, contributes, and depends and some new types related to agent based systems such as uses, creates, achieves, composed of, sends and receives. These new types were identified from concepts in *i**, Prometheus and JACK.

The process to identify the relations types was to find all relations between elements and then define the types based on semantic of each relation.

The reference model considers the main elements used by our case studies. For instance, we did not consider softgoals in *i**, and different types of messages in JACK. Knowing the granularity of the elements to be linked is important to the consistency of traceability and strongly influences the cost-benefits of the traceability effort. Our reference model reflects what we believe to be the most important elements and the right granularity to be traced. The set of relations types that we have proposed are considered the main ones and associate the main (important) elements in the documents of our concerned. In addition, they are important to support certain activities such as impact analysis, verification and validation.

We present below descriptions of these different types of traceability relations and illustrate these relations with some examples. Appendices F and G present a complete set of examples for all the different types of traceability relations.

3.3.1 Traceability Relations between i* and Prometheus

Tables 3.1 and 3.2 present the different types of traceability relations for the main types of elements in (a) *i** SD model and Prometheus models, and (b) *i** SR model and Prometheus models, respectively. In Tables 3.1 and 3.2, apart from overlaps relations that are bi-directional, the direction of a relation is represented from a row [i] to a column [j] (e.g. "Prometheus role contributes to SD goal"). We do not consider *i** soft goals in Tables 3.1 and 3.2 since soft goals are concerned with non-functional aspects of a system while elements in Prometheus are concerned with functional aspects of a system. We define below the various types of traceability relations and give some examples from the perspective of some associated artefacts.

i* **Prometheus**	**SD Goal**	**SD Resource**	**SD Task**	**Actor**
Goal	Overlaps	---	Overlaps	Is Dependent
Role	Contributes to	Uses	Contributes to	Contributes to
Agent	Achieves	Uses	Achieves	Overlaps
Capability	Contributes to	Uses	Contributes to	Compose
Plan	Achieves	Uses	Achieves	Created by
Percept	---	Overlaps	---	---
Data	Contributes to	---	Contributes to	Is Used
Scenario	Depends on	Composed of	Depends on	Is Dependent
Message	---	Overlaps	---	---

Table 3.1 Relations between Prometheus and *i SD**

i* Prometheus	SR Goal	SR Resource		SR Task
Goal	Overlaps	---		Overlaps
Role	Achieves	Uses	Creates	Achieves
Agent	Achieves	Uses	Creates	Achieves
Capability	Contributed by	Uses	Creates	Achieves
Plan	Achieves	Uses	Creates	Achieves
Action	---	---		Overlaps
Data	Used by	Overlaps		Used by
Scenario	Composed of	Uses	Creates	Composed of

Table 3.2 Relations between Prometheus and *iSR elements**

- Overlaps – in this type of relation, an element e1 overlaps with an element e2 (an element e2 overlaps with an element e1), if e1 and e2 refer to elements with common aspects of the agent software development. As shown in Tables 3.1 and 3.2, an *overlaps* relation may hold between a) goal in Prometheus and SD goal in *i**; b) goal in Prometheus and a SD task; c) an agent in Prometheus and an actor in *i** d) a percept in Prometheus and a SD resource in *i**; d) a message in Prometheus and a SD resource in *i**; e) a goal in Prometheus and a SR goal in *i**; f) a goal in Prometheus and a SR task in *i**; g) an action in Prometheus and a SR task in *i**; h) a data in Prometheus and a SR Resource in *i**.

In order to illustrate, consider the situation in which a goal g_1 in Prometheus has an *overlaps* traceability relation with a SD goal g_2 in *i**, if the name of the goal g_1 is a synonym of the name of the goal g_2 and the number of sub-elements of goal g_1 that is similar to the sub-goals and sub-tasks of goal g_2 is greater than a threshold (e.g. 40%), or the number of sub-elements of the Prometheus goal g_1 that is similar to the sub-goals and sub-tasks of goal g_2 is greater than a threshold (e.g. 60%). For instance, *Browse Book* SD goal in *i** has a synonym name to *Browse book* goal in Prometheus (see Figure 3.17). *Browse Book* SD goal is decomposed on *Browse By Special Offer, Browse By BestSeller, Browse By Category, Browse by New Releases* sub-goals and *Browse book* goal is decomposed on *Browse By category*, *Browse by new release, Browse by bestseller*, and *Browse by special offer* sub-tasks. The degree of similarity between the sub-elements of *Browse Book* SD goal and *Browse Book* Prometheus goal is equal to 100% because *Browse By Special Offer, Browse By*

BestSeller, Browse By Category, Browse by New Releases sub-tasks are synonyms *to Browse by category, Browse by new release, Browse by bestseller, Browse by special offer* sub-goals, respectively. Therefore, there is an overlap traceability relation between *Browse Book* Prometheus goal and *Browse book* SD goal.

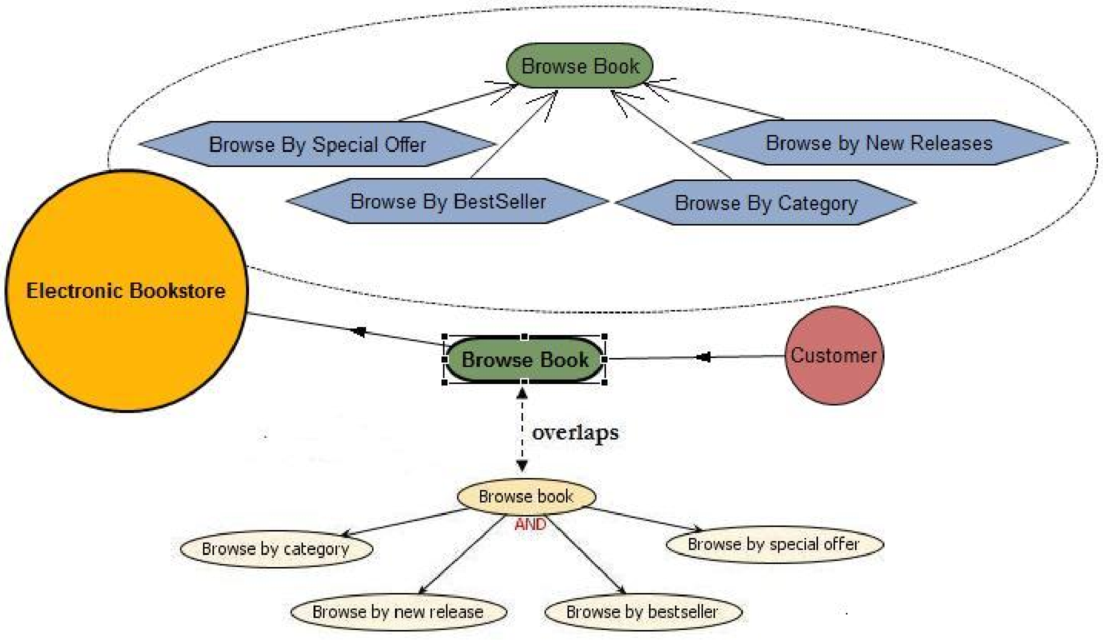


Figure 3.17 Prometheus Goal vs. SD Goal overlaps traceability relation

A data d_1 in Prometheus has an *overlaps* traceability relation with a SR resource r_1 in *i** when the name of the data d_1 is synonym with the name of SR resource. For instance, Customer Order SR resource has a synonym name with the of the Customer Order data in Prometheus (see Figure 3.18) Therefore, there is an overlaps traceability relation between Customer Order SR resource and Customer Order data in Prometheus.

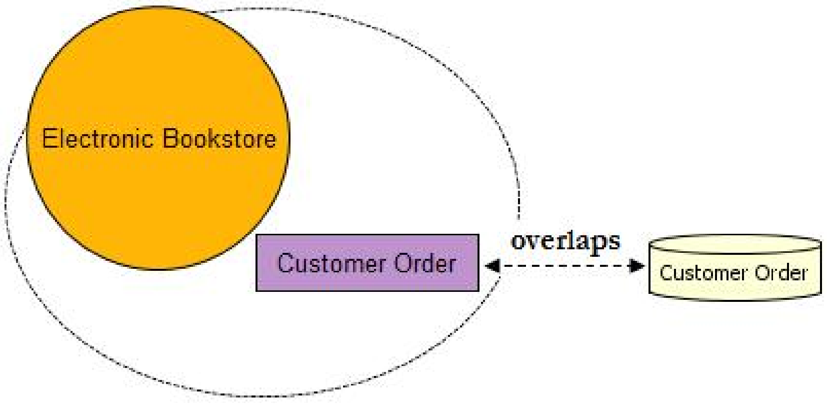


Figure 3.18 Prometheus Data vs. SR Resource overlaps traceability relation

- Contributes (Contributed by) - in this type of relation, an element e1 contributes to an element e2, if e1 helps to achieve or realise another element e2. As shown in Tables 3.1 and 3.2, a *contributes* relation may hold between a) role in Prometheus and a SD goal in *i**; b) a role in Prometheus and a SD Task in *i**; c) a role in Prometheus and actor in *i**; d) a capability in Prometheus and a SD goal in *i**; e) a capability in Prometheus and a SD task in *i**; f) a data in Prometheus and a SD goal in *i**; g) a data in Prometheus and a SD task in *i**; h) a capability in Prometheus and a SR goal in *i**.

In order to illustrate, consider the situation in which a data d_1 in Prometheus has a *contributes* traceability relation with SD goal g_1 in *i** when some of the sub-resources of SD goal g_1 has an *overlaps* traceability relation with data d_1. For instance, Customer Order SR resource is sub-resource of Buy Book SD goal and Customer Order SR Resource has an overlaps traceability relation with Customer Order Prometheus data (see Figure 3.19). Therefore, there is a contributes traceability relation between Customer Order data in Prometheus and Buy Book SD goal in *i**.

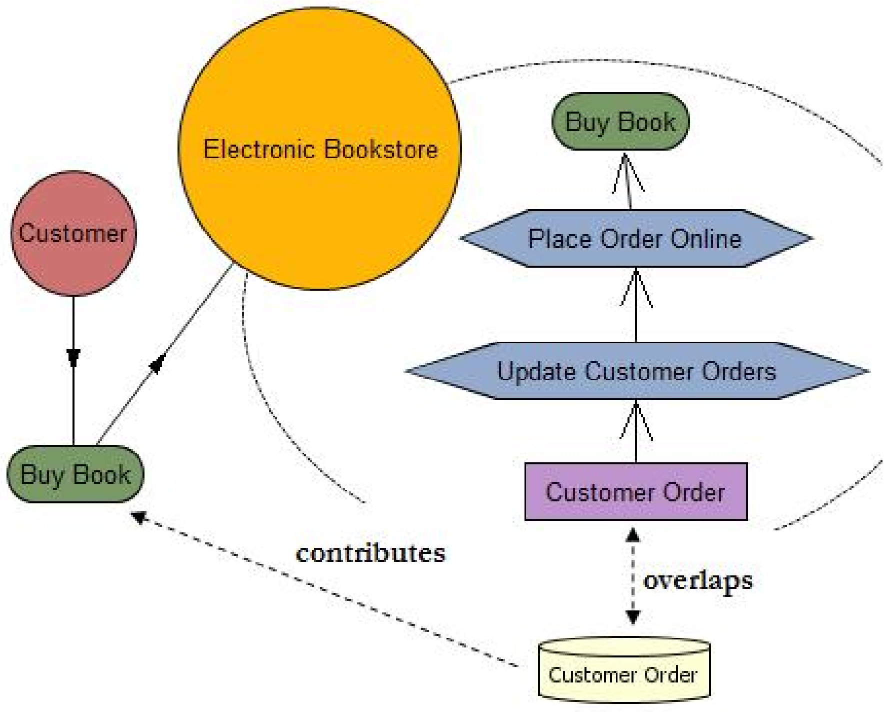


Figure 3.19 Prometheus Data vs. SD Goal contributes traceability relation

A data d_1 in Prometheus has a *contributes* traceability relation with SD Task t_1 in *i** when some of the sub-resources of SD Goal g_1 has an *overlaps* traceability relation with data d_1. For instance, balances SD resource in *i** is a sub-resource of Process Withdraw SD task and balances SD resource has an overlaps traceability relation with balances data in Prometheus (see Figure 3.20). Therefore, there is a contributes traceability relation between balances data in Prometheus and Process Withdraw SD task in *i**.

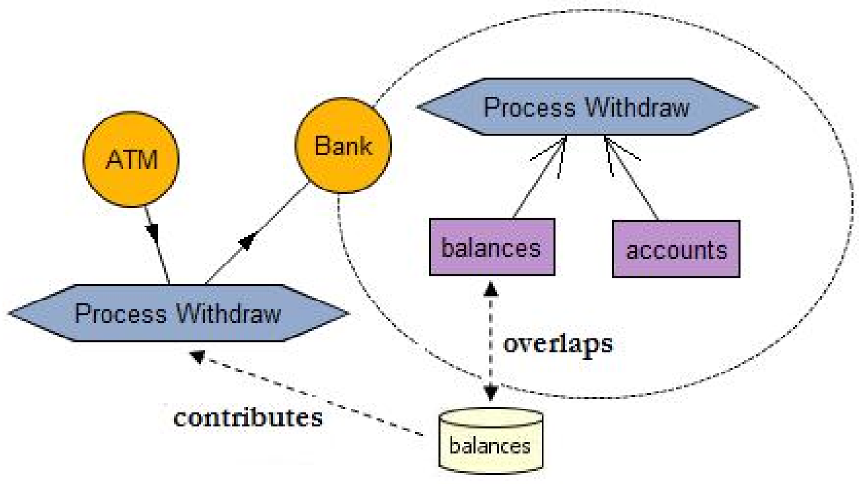


Figure 3.20 Prometheus Data vs. SD Task contributes traceability relation

- Uses (Used by) - in this type of relation, an element e1 uses an element e2, if e1 requires the existence of e2 in order to achieve its objective. As shown in Tables 3.1 and 3.2, a *contributes* relation may hold between a) a role in Prometheus and a SD resource in *i**; b) an agent in Prometheus and a SD resource in *i**; c) a capability in Prometheus and a SD resource in *i**; d) a plan in Prometheus and a SD resource in *i**; e) a data in Prometheus and an actor in *i**; f) a role in Prometheus and a SR resource in *i**; g) an agent in Prometheus and a SR goal in *i**; h) a capability in Prometheus and a SR resource in *i**; i) a plan in Prometheus and a SR goal in *i**; j) a data in Prometheus and a SR resource in *i**.

In order to illustrate, consider the situation in which a plan p_1 in Prometheus has an *uses* traceability relation with SD Resource r_1 when plan p_1 receives a message m_1 that has an *overlaps* traceability relation with resource r_1. For instance, RequestSlot plan in Prometheus receives AircraftEvent message (see Figure 3.21). AircraftEvent message in Prometheus has an overlaps traceability relation with Slot Allocated SD resource. Therefore, there is an uses traceability relation between RequestSlot plan and Slot Allocated SD resource.

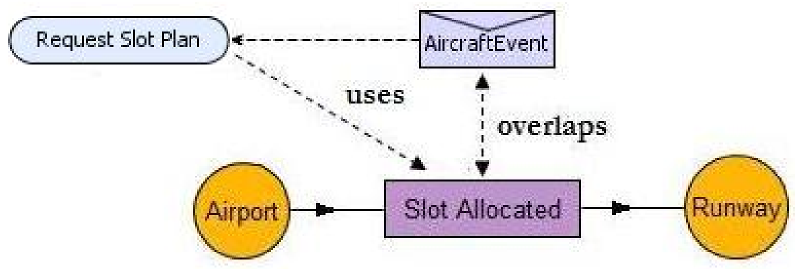


Figure 3.21 Prometheus Plan vs. SD Resource uses traceability relation

A plan p_1 in Prometheus has *uses* traceability relation with SR Resource r_1 in *i** when the plan p_1 reads a data d_1 that has an *overlaps* traceability relation with SR Resource r_1. For instance, *Monitor Aircraft* plan reads Landing Information data and *Landing Information* SR Resource has an *overlaps* traceability relation with *Landing Information* data in Prometheus (see Figure 3.22). Therefore, there is an *uses* traceability relation between *Monitor Aircraft* plan and *Landing Information* SR resource.

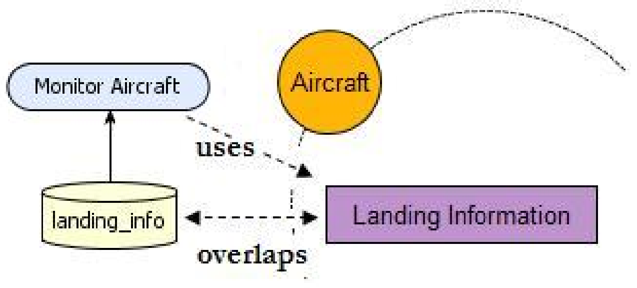


Figure 3.22 Prometheus Plan vs. SR Resource uses traceability relation

- Creates (Created by) - in this type of relation an element e1 creates an element e2, if e1 generates element e2. As shown in Tables 3.1 and 3.2, a *creates* relation may hold between a) a plan in Prometheus and an actor in *i**; b) role in Prometheus and SR resource in *i**; c) an agent in Prometheus and a SR resource in *i**; d) a capability in Prometheus and a SR resource in *i**; e) a plan in Prometheus and a SR resource in *i**; f) a scenario in Prometheus and a SR resource in *i**.

In order to illustrate, consider the situation in which a plan p_1 in Prometheus has *creates* traceability relation with a SR Resource r_1 in *i** when the plan p_1 writes a data d_1 that has an *overlaps* traceability relation with the SR Resource r_1. For instance, Assign Slot Plan writes landing_info data that has an overlaps traceability relation with Landing Information SR resource (see Figure 3.23). Therefore, there is a creates traceability relation between Assign Slot Plan plan in Prometheus and Landing Information SR resource.

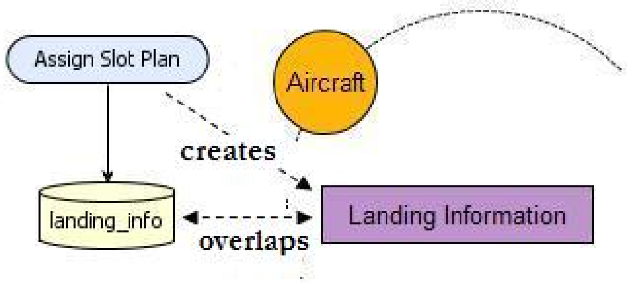
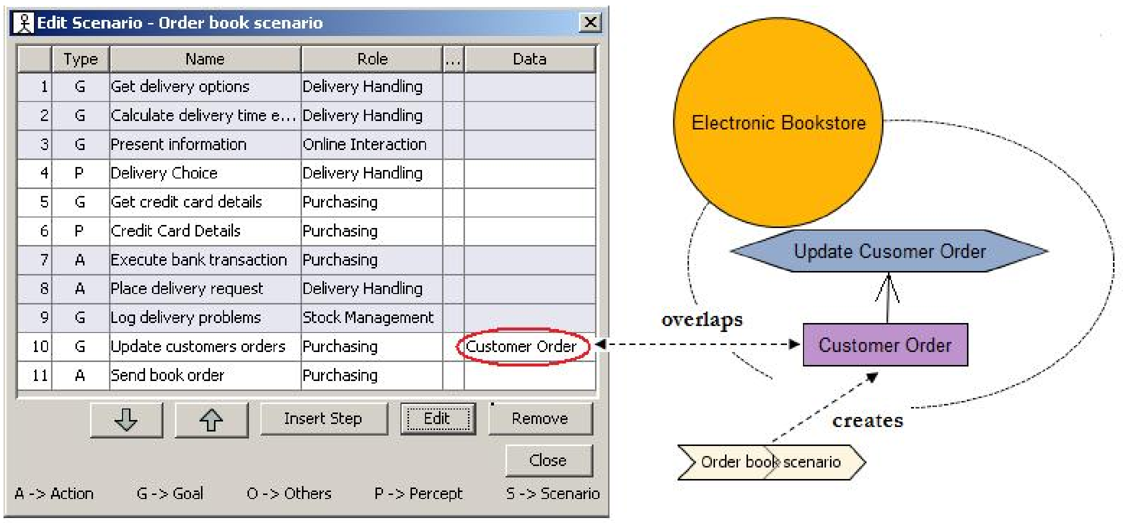


Figure 3.23 Prometheus Plan vs. SR Resource creates traceability relation

A scenario s_1 in Prometheus has a creates traceability relation with a SR resource r_1 in *i** when one of the steps of the scenario s_1 writes on data that has an *overlaps* traceability relation with resource r_1. For instance, Update customer orders step writes on Customer Order data that has an overlaps traceability relation with Customer Order SR resource (see Figure 3.24). Therefore, there is a creates traceability relation between Order book scenario and Customer Order SR resource.


Edit Scenario - Order book scenario

	Type	Name	Role	...	Data
1	G	Get delivery options	Delivery Handling		
2	G	Calculate delivery time e...	Delivery Handling		
3	G	Present information	Online Interaction		
4	P	Delivery Choice	Delivery Handling		
5	G	Get credit card details	Purchasing		
6	P	Credit Card Details	Purchasing		
7	A	Execute bank transaction	Purchasing		
8	A	Place delivery request	Delivery Handling		
9	G	Log delivery problems	Stock Management		
10	G	Update customers orders	Purchasing		Customer Order
11	A	Send book order	Purchasing		

A -> Action G -> Goal O -> Others P -> Percept S -> Scenario

Figure 3.24 Prometheus Scenario vs. SR Resource creates traceability relation

- Achieves (Achieved by) - in this type of relation an element e1 achieves an element e2, if e1 meets the expectations and needs of e2. As shown in Tables 3.1 and 3.2, an *achieves* relation may hold between a) an agent in Prometheus and a SD goal in *i**; b) an agent in Prometheus and a SD task in *i**; c) a plan in Prometheus and a SD goal in *i**; d) a plan in Prometheus and a SD task in *i**; e) a role in Prometheus and a SR goal in *i**; f) a role in Prometheus and a SR task in *i**; g) an agent in Prometheus and a SR task in *i**; h) a capability in Prometheus and a SR task in *i**; i) a plan in Prometheus and and a SR task in *i**.

In order to illustrate, consider the situation in which an agent a_1 in Prometheus has an achieves traceability relation with a SD Goal g_1 when the Prometheus agent a_1 achieves a goal g_2 and the goal g_2 has an overlaps traceability relation with the goal g_1. For instance, Stock Manager agent achieves Browse book goal in Prometheus and Browse book goal in Prometheus has an overlaps traceability relation with Browse Book SD goal (see Figure 3.25). Therefore, there is a achieves traceability relation between Stock Manager agent in Prometheus and Browse Book SD goal in *i**.

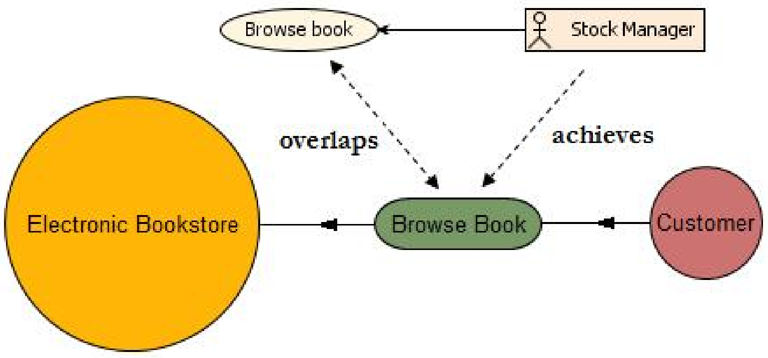


Figure 3.25 Prometheus Agent vs. SD Goal achieves traceability relation

In order to illustrate, a plan p_1 in Prometheus has an *achieves* traceability relation with a SR Task t_1 in *i** when the plan p_1 achieves a goal g_1 that has *overlaps* traceability relation with the SR Task t_1. For instance, Initiate Approach plan achieves Initiate Aircraft Approach goal and Initiate Aircraft Approach goal has an overlaps traceability relation with Initiate Approach SR task (see Figure 3.26). Therefore, there is an achieves traceability relation between Initiate Approach plan in Prometheus and Initiate Approach SR task

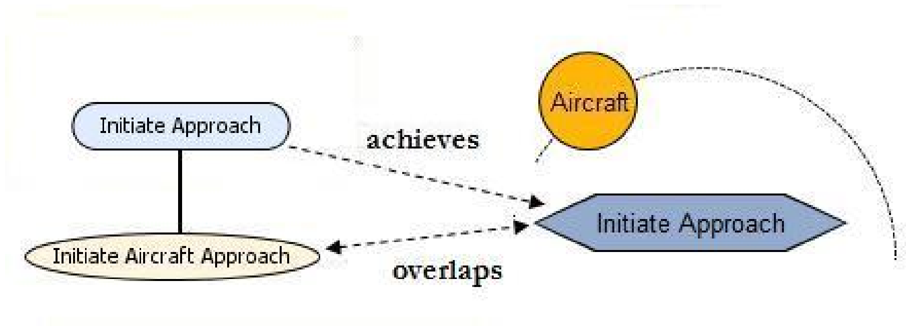


Figure 3.26 Prometheus Plan vs. SR Task achieves traceability relation

- Depends on (Is Dependent) - in this type of relation an element e1 depends on an element e2, if the existence of e1 relies on the existence of e2, or if changes in e2 have to be reflected in e1. As shown in Tables 3.1 and 3.2, a *depends* relation may hold between a) a goal in Prometheus and an actor in *i**; b) a scenario in Prometheus and a SD goal in *i**; c) a scenario in Prometheus and an actor in *i**.

In order to illustrate, consider the situation in which a goal g_1 in Prometheus has a *depends on* traceability relation with an actor a_1 in *i** when the goal g_1 has an *overlaps* traceability relation with a goal g_2 in *i** and the actor depends on that goal g_2. For instance, Browse Book SD goal has an *overlaps* traceability relation with Browse book goal in Prometheus (see Figure 3.27). Therefore, a *depends on* traceability relation is created between the Customer actor in *i** and browse book goal in Prometheus.

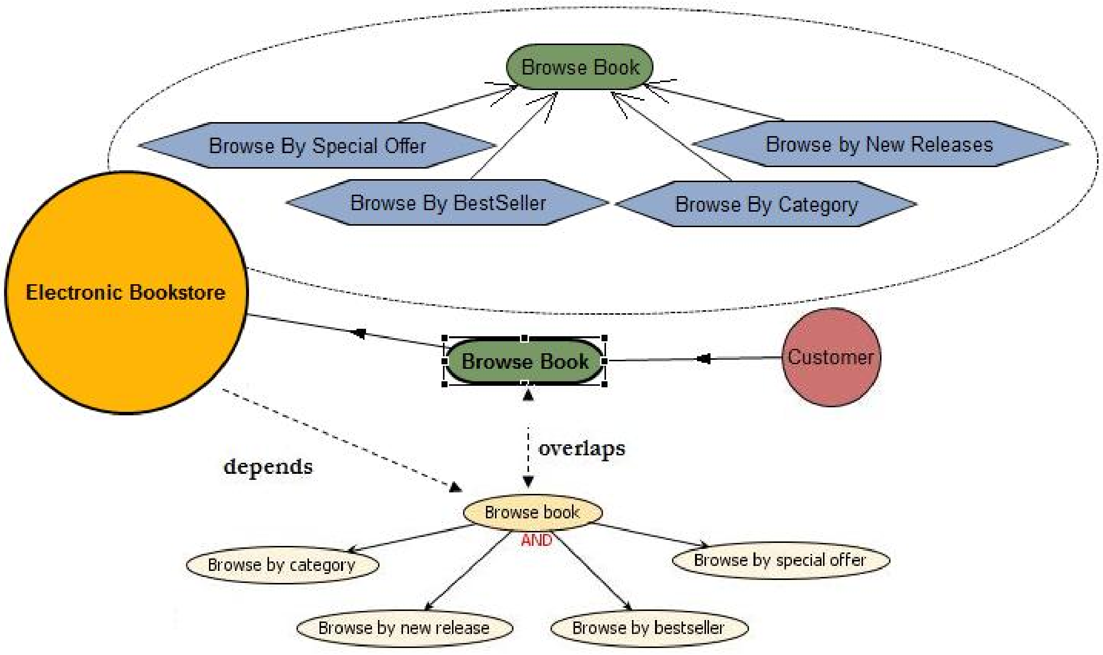


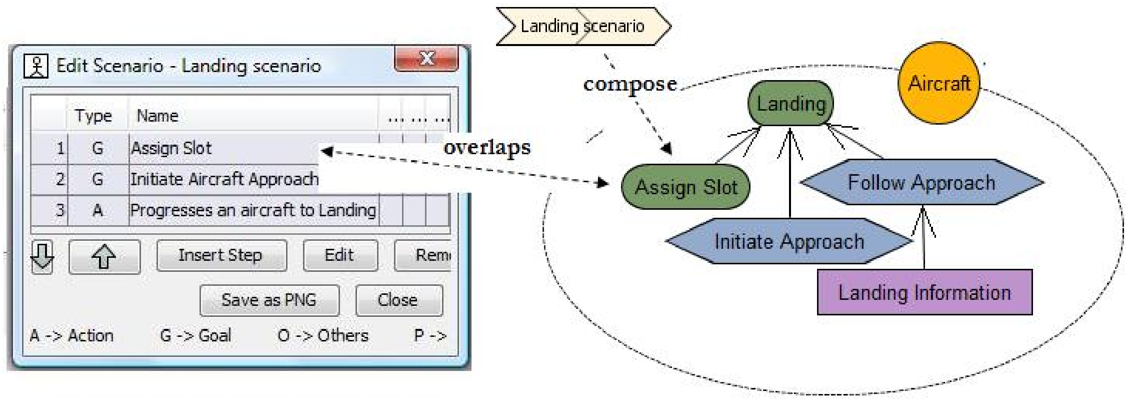
Figure 3.27 Prometheus Goal vs. Actor depends on traceability relation

A scenario s_1 in Prometheus has a *depends on* traceability relation with a SD Goal g_1 in *i** when the number of sub-elements of the goal g_1 that has an *overlaps* traceability relation with the steps of the scenario s_1 is greater than a threshold (e.g. 80%) and the name of the scenario is synonyms to the name of the SD goal. For instance, Order book scenario (see Figure 3.7) is composed of the following steps: Get delivery options goal, Calculate delivery time estimates goal, Present information goal, Delivery Choice percept, Get credit card details goal, Credit Card Details percept, Execute bank transaction action, Place delivery request action, Log delivery problems goal, Update customers orders goal, Send book order action . Buy Book SD goal (see Figure 3.3) is decomposed in Place Order Online SR task, Place Order By Phone SR task, Send Book Order Confirmation SR task, Update Customer Orders SR task, Customer Order SR task, Make Payment SR task, Perform Bank Transaction SR task, Transaction Accepted SR task, Transaction Rejected SR task, Obtain Credit Card Details SR task, Delivery Handling SR task, Fill Pending Order SR task, Organize Delivery SR task, Log Outgoing Delivery SR task, Place Delivery Request SR task, Compute Delivery Time Estimates SR task, and Obtain Delivery Options SR task, Postal DB SR resource, Courier DB SR resource. Get delivery option goal in Prometheus has an overlaps relation with Obtain Delivery Options SR task, Calculate delivery time has an overlaps traceability relation with estimates goal and

Compute Delivery Time Estimates SR task, Delivery choice percept has an overlaps traceability relation with Delivery Choice SD resource, Get credit card details goal has an overlaps traceability with Obtain Credit Card Details SR task, Credit Card Details percept has an overlaps traceability Credit Card Details SD resource, Execute bank transaction action has an overlaps traceability relation with Perform Bank Transaction SR task, Place delivery request action has an overlaps traceability relation with Place Delivery Request SR task, Log delivery problems goal has an overlaps traceability relation with Log Delivery Problems SR task, Update customer orders goal has an overlaps traceability relation with Update Customer Orders SR task, Send book order action has an overlaps traceability relation with Send Book Order Confirmation SR task. Therefore, there is a *depends* traceability relation between Order book scenario and Buy Book SD goal since 90,90% of steps of the Order book scenario has an overlaps traceability relation with sub-elements of the Buy Book SD goal and Order book and Buy Book are synonyms

- Composed of - in this type of relation an element e1 is composed of an element e2, if e1 is a complex element formed by element e2. As shown in Tables 3.1 and 3.2, a *composed of* relation may hold between a) a capability in Prometheus and an actor in *i**; b) a scenario in Prometheus and a SD resource; c) a scenario in Prometheus and a SR goal in *i**; d) a scenario in Prometheus and a SR task in *i**; e) a scenario in Prometheus and a SR goal in *i**; f) a scenario in Prometheus and a SR task in *i**.

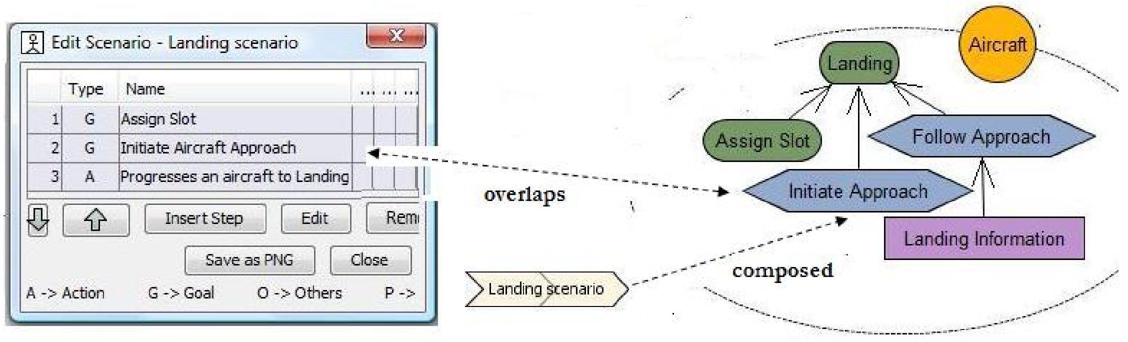
In order to illustrate, consider the situation in which a scenario s_1 in Prometheus has a *composed* traceability relation with a SR Goal g_1 in *i** when a step of the scenario s_1 and the goal g_1 has an overlaps traceability relation. For instance, Assign Slot SR goal has an overlaps traceability with Assign Slot step of the Landing scenario (see Figure 3.28). Therefore, there is a composed traceability relation between Landing scenario and Assign Slot SR goal.


Edit Scenario - Landing scenario

	Type	Name
1	G	Assign Slot
2	G	Initiate Aircraft Approach
3	A	Progresses an aircraft to Landing

Figure 3.28 Prometheus Scenario vs. SR Goal compose traceability relation

A scenario s_1 in Prometheus has a *composed* traceability relation with a SR task t_1 in *i** when a step of the scenario s_1 and the task t_1 has an overlaps traceability relation. For instance, Initiate Approach SR task has an overlaps traceability relation with Initiate Aircraft Approach step of the Landing scenario (see Figure 3.29). Therefore, there is a composed traceability relation between Landing scenario and Initiate Aircraft SR task.


Edit Scenario - Landing scenario

	Type	Name
1	G	Assign Slot
2	G	Initiate Aircraft Approach
3	A	Progresses an aircraft to Landing

Figure 3.29 Prometheus Scenario vs. SR Task composed traceability relation

3.2.2 Traceability Relations between Prometheus and JACK

Tables 3.3 and 3.4 present different types of traceability relations for the main types of elements in Prometheus and JACK models. As in the case of Tables 3.1 and 3.2, in Tables 3.3 and 3.4, apart from overlaps relations that are bi-directional, the direction of a relation is represented from a row[i] to a column[j] (e.g. "An agent in JACK achieves a goal in Prometheus").

Prometheus JACK	Goal	Role	Agent	Capability
Agent	Achieves	Uses	Overlaps	Uses
Plan	Achieves	Is Used by	Is Used by	Is Used by
BeliefSet	---	Creates/Uses	Creates/Uses	Creates/Uses
Event	---	---	Is Sent by/ Is Received by	Is Sent by/ Is Received by

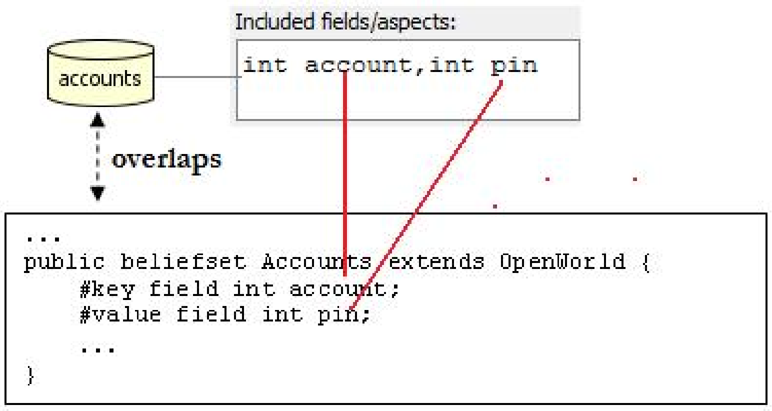
Table 3.3 Traceability Relations Types between Prometheus and JACK Artefacts

Prometheus JACK	Plan	Percept	Action	Message	Data
Agent	Uses	Uses	Creates	Send/Receives	Uses/Creates
Plan	Overlaps	Uses	Creates	Send/Receives	Uses/Creates
BeliefSet	Creates/Uses	---	---	---	Overlaps
Event	Send/Receives	---	---	Overlaps	---

Table 3.4 Traceability Relations Types between Prometheus and JACK Artefacts

- Overlaps – in this type of relation, an element e1 overlaps with an element e2 (an element e2 overlaps with an element e1), if e1 and e2 refer to common aspects of the agent software development. As shown in Tables 3.3 and 3.4, an *overlaps* relation may hold between a) an agent in JACK and an agent in Prometheus; b) a plan in JACK and a plan in Prometheus; c) a beliefSet in JACK and a data in Prometheus; an event in JACK and Prometheus message.

In order to illustrate, consider the situation in which a data in Prometheus and a beliefSet in JACK has an *overlaps* traceability relation when the name of the data is synonyms to the name of the beliefSet and if the name of the fields of the data and the beliefSet are similar (see Figure 3.30). For instance, the name of accounts data in Prometheus is synonyms to the name of Accounts beliefSet in JACK and the fields of the Accounts beliefSet (account and pin) are similar to the fields of the accounts data in Prometheus.

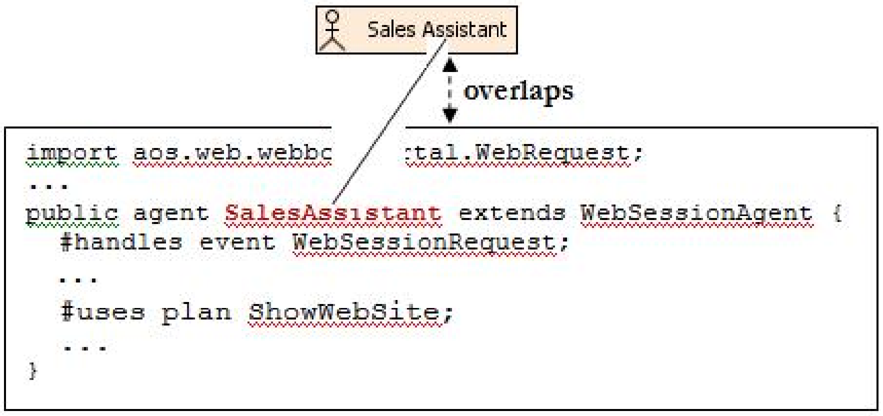


```
int account,int pin
```

```
...
public beliefset Accounts extends OpenWorld {
    #key field int account;
    #value field int pin;
    ...
}
```

Figure 3.30 JACK BeliefSet vs. Prometheus Data overlaps traceability relation

An agent a_1 in JACK has an *overlaps* traceability relation with an agent a_2 in Prometheus when the name of the agent a_1 in JACK is synonyms to the name of the agent a_2 in Prometheus. For instance, Sales Assistant agent in Prometheus has *synonyms* name to SalesAssistant agent in JACK (see Figure 3.31). Therefore, there is an overlaps traceability relation between Sales Assistant agent in Prometheus and SalesAssistant agent in JACK.

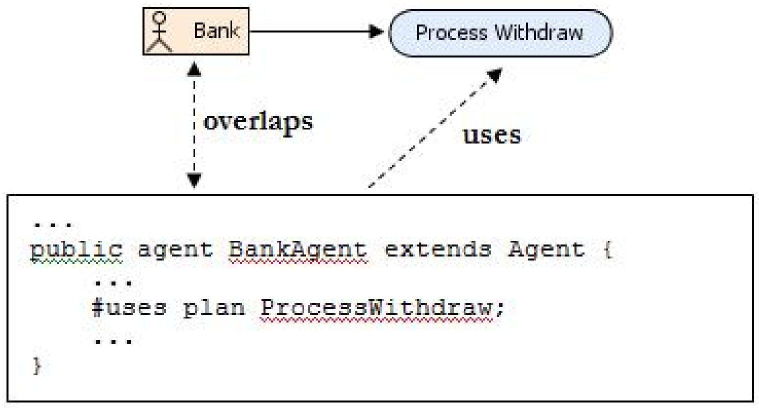


```
import aos.web.webbc    :tal.WebRequest;
...
public agent SalesAssistant extends WebSessionAgent {
  #handles event WebSessionRequest;
  ...
  #uses plan ShowWebSite;
  ...
}
```

Figure 3.31 JACK Agent vs. Prometheus Agent overlaps traceability relation

- Uses (Used by) - in this type of relation, an element e1 uses an element e2, if e1 requires the existence of e2 in order to achieve its objective. As shown in Tables 3.3 and 3.4, an *uses* relation may hold between a) an agent in JACK and a role in Prometheus; b) a plan in JACK and a role in Prometheus; c) a plan in JACK and a capability in Prometheus; d) a beliefSet in JACK and a role in Prometheus; e) a beliefSet in JACK and a capability in Prometheus.

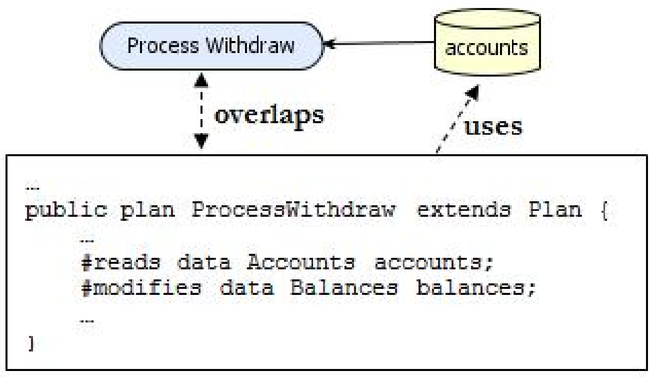
In order to illustrate, consider the situation in which an agent a_1 in JACK has *uses* traceability relation with a plan p_1 in Prometheus when there is an *overlap* traceability relation between the JACK agent a_1 and a Prometheus agent a_2 in Prometheus and the agent a_2 includes the plan p_1 in Prometheus. For instance, BankAgent agent in JACK has an overlaps traceability relation with Bank agent in Prometheus and Bank agent in Prometheus uses Process Withdraw plan (see Figure 3.32). Therefore, there is an uses traceability relation between BankAgent in JACK and Process Withdraw plan in Prometheus.



```
...
public agent BankAgent extends Agent {
    ...
    #uses plan ProcessWithdraw;
    ...
}
```

Figure 3.32 JACK Agent vs. Prometheus Plan uses traceability relation

A plan p_1 in JACK has *uses* traceability relation with a data d_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and a plan p_2 in Prometheus and the Prometheus plan p_2 *uses* the data d_1. For instance, *Execute Advanced Search* plan in Prometheus has an *overlaps* traceability relation with *ExecuteAdvancedSearch* plan in JACK and *Execute Advanced Search* plan in Prometheus reads or modifies *BooksDB* data in Prometheus. Therefore, there is an *uses* traceability relation between ExecuteAdvancedSearch plan in JACK and BooksDB data in Prometheus. For instance, Process Withdraw plan in Prometheus and ProcessWithdraw plan in JACK has an overlaps traceability relation and Process Withdraw plan in Prometheus reads accounts data (see Figure 3.33). Therefore, there is an uses traceability relation between ProcessWithdraw plan in JACK and accounts data in Prometheus.

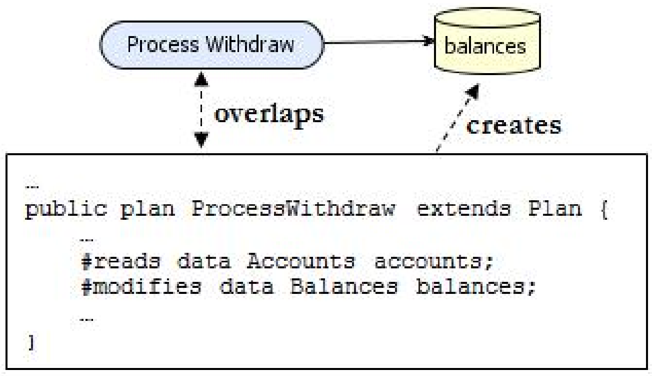


```
...
public plan ProcessWithdraw  extends Plan {
    ...
    #reads data Accounts accounts;
    #modifies data Balances balances;
    ...
}
```

Figure 3.33 JACK Plan vs. Prometheus Data uses traceability relation

- Creates (Created by) - in this type of relation an element e1 creates an element e2, if e1 generates element e2. As shown in Tables 3.3 and 3.4, a *creates* relation may hold between a) a plan in Prometheus and an actor in *i**; b) role in Prometheus and SR resource in *i**; c) an agent in Prometheus and a SR resource in *i**; d) a capability in Prometheus and a SR resource in *i**; e) a plan in Prometheus and a SR resource in *i**; f) a scenario in Prometheus and a SR resource in *i**.

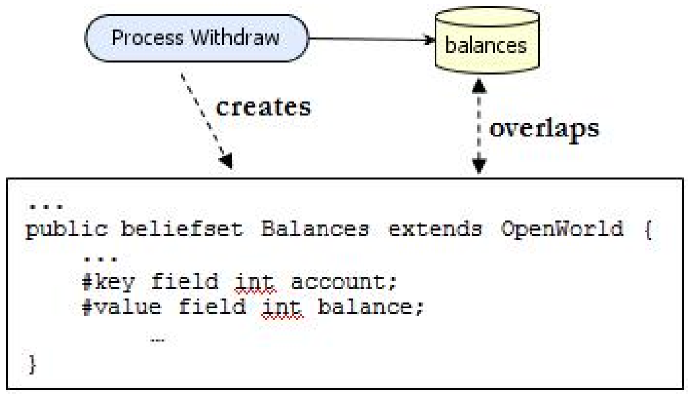
In order to illustrate, consider the situation in which a plan p_1 in JACK has *creates* traceability relation with a data d_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and a plan p_2 in Prometheus and the Prometheus plan p_2 *creates* the data d_1. For instance, Process Withdraw plan in Prometheus has an overlaps traceability relation with ProcessWithdraw plan in JACK and ProcessWithdraw plan writes on balances data (see Figure 3.34). Therefore, there is a creates traceability relation between ProcessWithdraw plan in JACK and balances data in Prometheus.



```
...
public plan ProcessWithdraw  extends Plan {
    ...
    #reads data Accounts accounts;
    #modifies data Balances balances;
    ...
}
```

Figure 3.34 JACK Plan vs. Prometheus Data creates traceability relation

A plan p_1 in Prometheus has *creates* traceability relation with a beliefSet b_1 in JACK when there is an *overlaps* traceability relation between the beliefSet b_1 in JACK and a data d_1 in Prometheus and the plan p_1 in Prometheus writes on the data d_1. For instance, Process Withdraw plan in Prometheus writes on balances data and balances data in Prometheus has an overlaps traceability relation with Balances beliefSet in JACK (see Figure 3.35). Therefore, there is a creates traceability relation between Process Withdraw plan in Prometheus and Balances beliefSet in JACK.

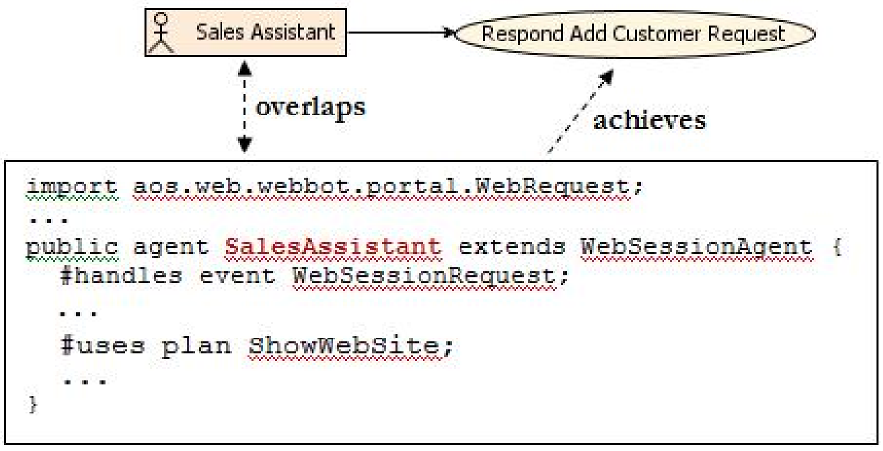


```
...
public beliefset Balances extends OpenWorld {
    ...
    #key field int account;
    #value field int balance;
        ...
}
```

Figure 3.35 JACK BeliefSet vs. Prometheus Plan creates traceability relation

- Achieves (Achieved by) - in this type of relation an element e1 achieves an element e2, if e1 meets the expectations and needs of e2. As shown in Tables 3.3 and 3.4, a *achieves* relation may hold between a) an agent in JACK and a goal in Prometheus; b) a plan in JACK and a goal in Prometheus.

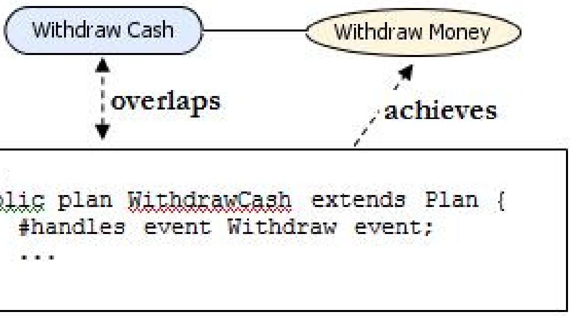
In order to illustrate, consider the situation in which an agent a_1 in JACK has an *achieves* traceability relation with a goal g_1 in Prometheus when there is *overlap* traceability relation between the JACK agent a_1 and an agent in Prometheus a_2 and the goal g_1 is one of the goals that the Prometheus agent a_2 achieves. For instance, Sales Assistant agent in Prometheus has an *overlaps* traceability relation with the SalesAssistant agent in JACK and the Sales Assistant agent in Prometheus *achieves* the *Respond Add Customer Request* goal in Prometheus (see Figure 3.36). Therefore, there is an *achieves* relation between the *SalesAssistant* agent in JACK and the *Respond Add Customer Request* goal in Prometheus.



```
import aos.web.webbot.portal.WebRequest;
...
public agent SalesAssistant extends WebSessionAgent {
  #handles event WebSessionRequest;
  ...
  #uses plan ShowWebSite;
  ...
}
```

Figure 3.36 JACK Agent vs. Prometheus Goal achieves traceability relation

A plan p_1 in JACK has an *achieves* traceability relation with a goal g_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and the plan p_2 in Prometheus and the Prometheus plan p_2 *achieves* the goal g_1. For instance, Withdraw Cash plan in Prometheus has an overlaps traceability relation with WithdrawCash plan in JACK and Withdraw Cash plan achieves Withdraw Money goal in Prometheus (see Figure 3.37). Therefore, there is an achieves traceability relation between WithdrawCash plan in JACK and Withdraw Money goal in Prometheus.

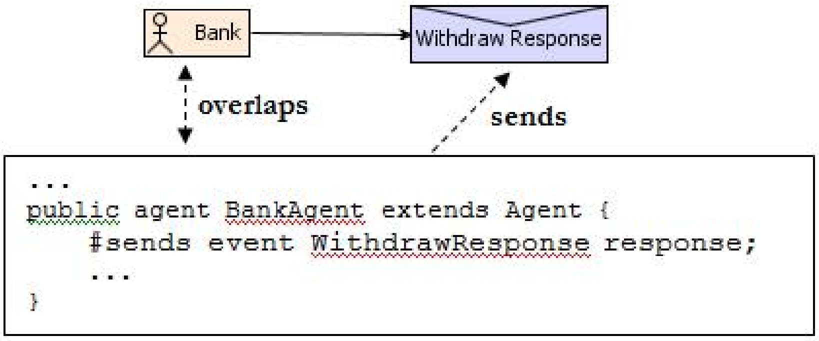


```
...
public plan WithdrawCash extends Plan {
    #handles event Withdraw event;
    ...
}
```

Figure 3.37 JACK Plan vs. Prometheus Goal achieves traceability relation

- Sends (Is Sent by) – As shown in Tables 3.3 and 3.4, a sends relation may hold between a) an event in JACK and an agent in Prometheus; b) an event in JACK and a capability in Prometheus; c) an agent in JACK and a message in Prometheus.

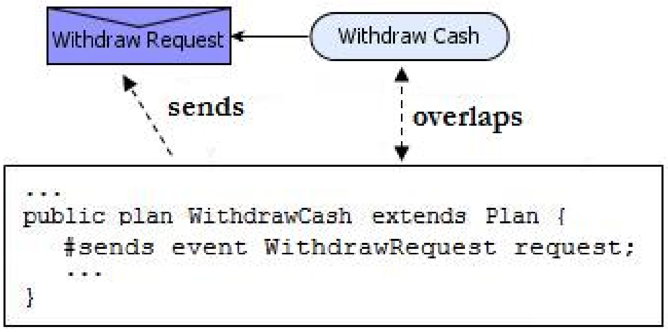
In order to illustrate, consider the situation in which an agent ag_1 in JACK has *sends* traceability relation with a message m_1 in Prometheus when there is an *overlaps* traceability relation between the agent ag_1 in JACK and an agent ag_2 in Prometheus and the Prometheus agent ag_2 in Prometheus *sends* the message m_1. For instance, Bank agent in Prometheus has an overlaps traceability relation with BankAgent in JACK and Bank agent in Prometheus sends Withdraw Response message (see Figure 3.38). Therefore, there is a sends traceability relation between BankAgent in JACK and Withdraw Response message in Prometheus.



```
...
public agent BankAgent extends Agent {
    #sends event WithdrawResponse response;
    ...
}
```

Figure 3.38 JACK Agent vs. Prometheus Message sends traceability relation

A plan p_1 in JACK has a *sends* traceability relation with a message m_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and a plan p_2 in Prometheus and the Prometheus plan p_2 *sends* the message m_1. For instance, Withdraw Cash plan in Prometheus has an overlaps traceability relation with WithdrawCash plan in JACK and Withdraw Cash plan in Prometheus sends Withdraw Request message (see Figure 3.39). Therefore, there is a sends traceability relation between WithdrawCash plan in JACK and Withdraw Request message in Prometheus.

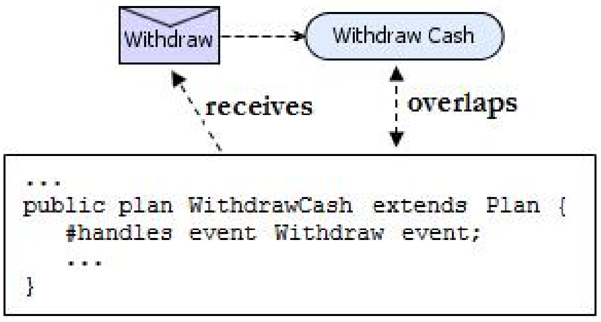


```
...
public plan WithdrawCash extends Plan {
   #sends event WithdrawRequest request;
   ...
}
```

Figure 3.39 JACK Plan vs. Prometheus Message sends traceability relation

- Receives (Is Received by) - As shown in Tables 3.3 and 3.4, a receives relation may hold between a) an event in JACK and an agent in Prometheus; b) an event in JACK and a capability in Prometheus; c) an agent in JACK and a message in Prometheus; d) an event in JACK and a plan in Prometheus.

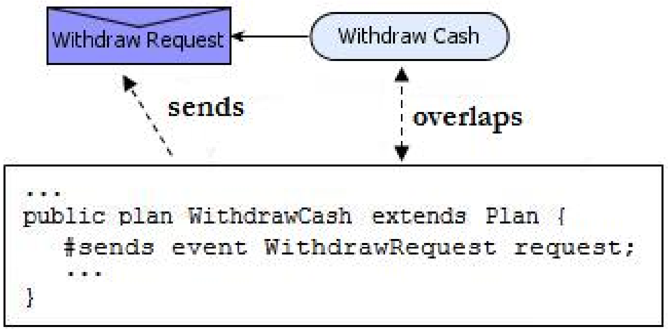
In order to illustrate, consider the situation in which a plan p_1 in JACK has *sends* traceability relation with a message m_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and a plan p_2 in Prometheus and the Prometheus plan p_2 *receives* the message m_1. For instance, Withdraw Cash plan in Prometheus has an overlaps traceability relation with WithdrawCash plan in JACK and Withdraw Cash plan in Prometheus receives Withdraw message (see Figure 3.40). Therefore, there is a receives traceability relation between WithdrawCash plan in JACK and Withdraw message in Prometheus.



```
...
public plan WithdrawCash extends Plan {
   #handles event Withdraw event;
   ...
}
```

Figure 3.40 JACK Plan vs. Prometheus Message receives traceability relation

A plan p_1 in JACK has a *sends* traceability relation with a message m_1 in Prometheus when there is an *overlaps* traceability relation between the plan p_1 in JACK and a plan p_2 in Prometheus and the Prometheus plan p_2 *sends* the message m_1. For instance, Withdraw Cash plan in Prometheus has an overlaps traceability relation with WithdrawCash plan in JACK and Withdraw Cash plan in Prometheus sends Withdraw Request message (see Figure 3.41). Therefore, there is a sends traceability relation between WithdrawCash plan in JACK and Withdraw Request message in Prometheus.



```
...
public plan WithdrawCash extends Plan {
   #sends event WithdrawRequest request;
   ...
}
```

Figure 3.41 JACK Plan vs. Prometheus Message sends traceability relation

3.4 Summary

This chapter described a traceability reference model for multi-agent systems developed using *i** framework, Prometheus methodology, and JACK language. It presented an overview of *i** framework, Prometheus methodology, and JACK language with its main documents, diagrams, and artefacts. It also presented various types of traceability relations that exist between these artefacts and gave examples of some of these traceability relations. Other examples of the various types of traceability relations are described in Appendices F and G.

Chapter 4 - Traceability Framework

In this chapter we present our rule-based traceability framework. to support (i) automatic generation of traceability relations between heterogeneous software models created during the development of multi-agent systems, and (ii) identification of missing elements in these various software models created during the development of multi-agent systems (completeness checking).

In section 4.1, we give an overview of the framework. We show the main components of the framework and explain the process of how traceability relations are created and missing information are identified by the framework. We describe how to create rules to generate traceability relations and to identify missing elements. We present a general template for our rules and explain in detail the different parts of a rule. We give examples of different types of rules to create traceability relations and identify missing elements for both *i** and Prometheus models and Prometheus and JACK models. In section 4.2, we describe different functions that we have developed to support the rules to perform completeness checking, to verify if names of elements in the models are synonyms, to compare similarities between elements in the models, and to manipulate elements in PDT, TAOME and JACK models. Finally, section 4.3 describes the prototype tool that we have developed to support our traceability framework.

4.1. Overview of the Framework

Our framework is intended to support automatic creation of traceability relations and completeness checking of multi-agent systems developed using models as primary engineering artefacts throughout the engineering lifecycle of multi-agent systems. In particular, our framework can support organisational *i** models created using TAOME4E tool (TAOME4E, 2008); Prometheus model created using PDT tool (PDT, 2010), (Padgham, et al., 2005); and code specifications implemented using JACK (Agent Oriented Software Limited, 2010).

We have adopted a rule-based approach due to the facts that (a) rules can automate and assist with decision making: if the condition of a rule is satisfied then an action is executed; and (b) rules can facilitate the generation of traceability relations and identification of missing elements: actions can be used to create traceability relations between software artefacts and to identify missing elements.

In order to support the heterogeneity of models and tools used during the software development life cycle we assume the models to be represented in XML. We have chosen XML as the basis of our approach due to several reasons: (a) XML has become the de facto language to support data interchange among heterogeneous tools and applications, (b) the existence of large number of applications that use XML to represent information internally or as a standard export format, and (c) to allow the use of XQuery as a standard way of expressing traceability rules. Moreover, our approach combines models in *i** and Prometheus and code in JACK, therefore, requires a common representation of these models.

We use an extended version of XQuery (XQuery, 2010) to represent the rules. XQuery is an XML-based query language that has been widely used for manipulating, retrieving, and interpreting information from XML documents. Apart from the embedded functions offered by XQuery, it is possible to add new functions. We have extended XQuery (a) to support representation of the consequence part of the rules, i.e. the actions to be taken when the conditions are satisfied; and (b) to support extra functions to (i) cover some of the traceability relations being proposed and (ii) completeness checking of the models. We describe in detail the list of functions created in Java to extend XQuery in the section 4.2.

Figure 4.1 presents an overview of our framework. As shown in the figure, initially, the models of our concern represented in their native format are generated using proprietary tools (e.g. TAOME4E (TAOME4E, 2008), PDT (PDT, 2010), (Padgham, et al., 2005), or any diagram editor tool. These models are translated into XML format (XML_based Models) by using a Model Translator component based on XML Schemas proposed for the models, whenever the tools used to create the models do not generate them directly in XML.

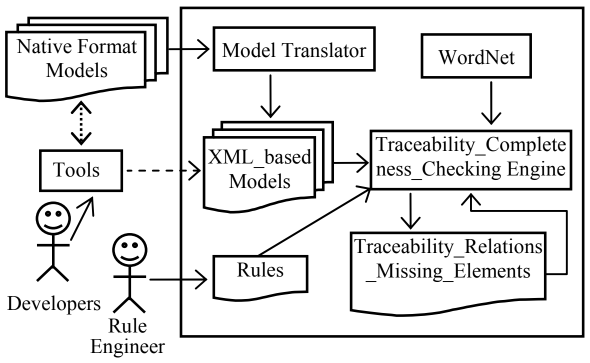


Figure 4.1: Overview of traceability framework

The XML based models and rules are used as inputs to the Traceability_Completeness_Checking Engine component to generate traceability relations between the models and to identify missing elements based on the rules. The engine also uses WordNet (WordNet, 2010) to support the identification of synonyms between the names of elements in the models. The WordNet is important component because in general naming conventions can change from high-level goal model (e.g. *i**) to low-level representations (e.g. JACK code).

As an example of the use of WordNet, consider the function isSynonym that we propose as an extension of XQuery (see Section 4.2 for more details). This function receives two terms as parameters and checks if these two terms are synonyms. Each parameter term is transformed into its root tem by using WordNet stemmer facility and is associated to a list of synonym terms based on WordNet. The isSynonym function verifies if at least one element in the list of synonyms of the first parameter term matches an element in the list of synonyms of the second term. In positive case, the terms in the parameters of the function are considered synonyms. For instance, suppose that isSynonyms function receives Arrange and Organize as parameters. The WordNet returns a list of synonyms to Organize (e.g. {arrange, systematize adapt, adjust, be responsible for, catalogue, classify, codify,...}) and a list of synonyms to Arrange (e.g. align, array, class, classify, file, fix up, form, group, line up, methodize, organize,...}) Arrange is synonyms to Organize and classify appears in the list of synonyms of Arrange and Organize. Therefore, WordNet considers Arrange and Organize as synonyms.

The isSynonym function also ignores code conventions. For instance, if in Prometheus model is defined an agent as Bank and in the implementation is defined an agent as JACKBank. The two names are considered synonyms because JACKBank contains Bank and Bank is the same word as Bank.

The traceability relations and identified missing elements are represented in an XML document (Traceability_Relations_Missing_Elements document). The use of a separated document to represent the traceability relations and missing elements is important to preserve the original models, to allow the use of these models by other applications and tools, and to allow the generated relations to be used to support the identification of other traceability relations that depend on the existence of previously identified relations (i.e. primitive and dependent relations, as described in Chapter 3). As shown in the Figure 4.1, the Traceability_Relations_Missing_Elements document is used as input to the Traceability_Completeness_Checking Engine component in order to support the generation of dependent traceability relations.

For instance, you can define a rule where the condition is that if a task in the Strategic Rationale (SR) model of an actor that represents a multi-agent system and a goal in Prometheus are similar, then a traceability relation can be created between the SR task in *i** and the goal in Prometheus. Figure 4.2 shows an example of how the Traceability_Completeness_Checking Engine component processes the rule between Arrange delivery goal in Prometheus and Organize Delivery SR task in *i**.

The rule checks first if the words used to give the name for the goal in Prometheus are synonyms to the words used to give the name for the SR task. Arrange is synonyms to Organize accordingly to WordNet dictionary and delivery and Delivery are the same word. Then the rule checks if the sub-goals of the *Arrange delivery* goal in Prometheus and the sub-goals and sub-tasks of the *Organize Delivery* SR task are similar. The similarity is calculated based on the names given to the elements and a threshold. In this particular case, the threshold is 50% that means that more than fifty percent of the names given to the sub-goals of *Arrange delivery* goal in Prometheus have to match (i.e. to be synonyms) to the names given to the sub-tasks or sub-goals of the *Organize Delivery* SR task in *i**. *Get delivery options* is synonyms to *Obtain Delivery Options* and *Calculate delivery time estimates* is synonyms to *Compute Delivery Time Estimates*. The percentage of sub-goals of the *Arrange delivery* goal that is similar to the sub-tasks or sub-goals of the *Organize Delivery* is 66.7% that is greater than the threshold of 50%. Therefore, an overlaps traceability relation is created between *Arrange delivery* Prometheus goal and *Organize Delivery* SR task (see Figure 4.2). Get and Obtain, Calculate and Compute are synonyms in the WordNet dictionary.

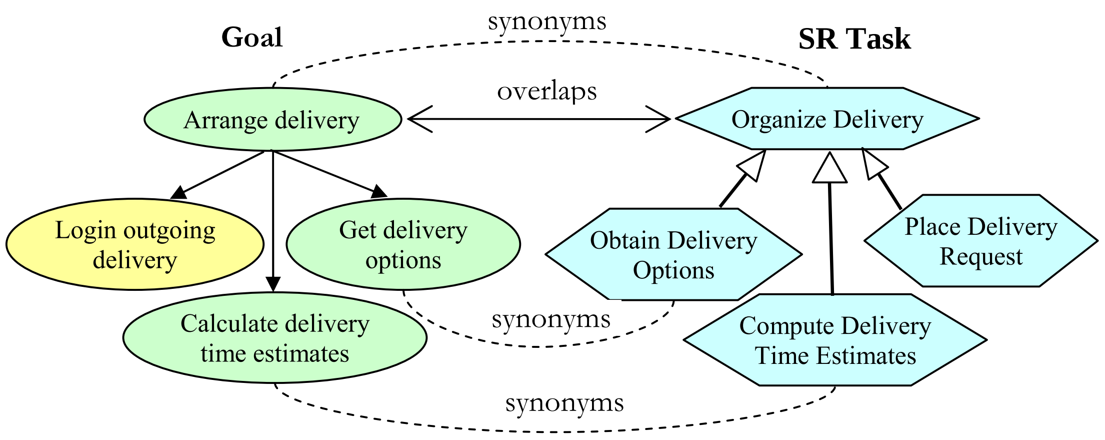


Figure 4.2 Example of the use of rule in our approach

The *Login outgoing delivery* sub-goal of the *Arrange delivery* goal does not match with any of sub-tasks or sub-goals of the *Organize Delivery* SR task in *i**. This information is represented as missing element.

It is worth noting that the tool does not enforce the analyst to complete the models after missing elements have been detected. The analyst can decide to use the information about missing elements to rectify the model or not to change the model. As in the example in Figure 4.2, the analyst could use the information about Login outgoing delivery that has been identified by the tool as missing and create a sub-task Login outgoing delivery of Organize Delivery SR Task. Similarly, the analyst could add a sub-goal Place Delivery Request to Arrange delivery goal, or decide that this information is not derived from the higher level model (i.e. *i** model) and, therefore, not add the sub-goal Place Delivery Request.

During software development, models are built representing different views on a software system. For instance, *i** represent organizational environment view of the system while goal modelling in Prometheus represent system specification of the multi-agent system. Some of goals in *i** might not be implemented by the multi-agent system specified in Prometheus. It can happen that some information detailed appear in Prometheus goal model that is not present in the *i** model. That is the reason that we do not enforce completeness. It is up the analyst to decide the level of detail that the models contain.

4.2 Traceability and Completeness Checking Rules

As described above, our framework is based on the use of rules to create traceability relations and to identify missing elements. Figure 4.3 presents a general template for our rules. As shown in Figure 4.3, a rule is composed of two main parts. The first part defines properties of the rule and the second part contains the XQuery code used to create traceability relations and to identify missing elements. We explain below the different parts in a rule.

Part 1: It consists of the rule identification and contains a unique identifier (*id*), a priority of the rule (*priority*) indicating if this is a primitive rule (priority = 1) or dependent rule (priority >1), the type of the rule (*type*), the type of the source element to be traced (*elemTypeA*), the type of the target element to be traced (*elemTypeB*), and a brief description of the rule (*description*). The priority of the rule is used to identify if the rule is primitive or dependent and to assist with the execution of the rules; i.e., rules with priority 1 are executed before rules with priority 2, and rules with priority 2 are executed before rules with priority 3, since rules with greater priority depend on the existence of the relations generated by rules with lower priority.

Part 2: The second part of the rule contains valid XQuery code and is composed of other sub-parts.

Sub-part 2.1 (DECLARE): It contains declaration of namespaces, variables, documents, and sequence of elements used by a rule, as described below. The declaration of the documents and sequence of elements are described by using XPath expressions.

- Namespace declarations – They are used to declare namespace of functions that have been implemented in Java to extend XQuery functionalities (see Section 4.2). For instance, the following statement in XQuery declares that all functions defined in the java:retratos.XQueryPDTFunctions class can be used by of the XQuery code (e.g. getPDTFileName).

    - declare namespace pdt = "java:retratos.XQueryPDTFunctions";

- Variable and Document declarations – They define a sequence of elements that are used by the rule and the names of the source and target documents to be compared by a rule. The following statements in XQuery show examples of variable declarations. In the first statement, the $prometheusDoc variable is assigned to the value of the name of a document that is returned by the getPDTFileName function in Java. In the second statement, the $prometheusGoals variable is assigned to the value returned after executing the XPath expression. More specifically, the $prometheusGoals variable is assigned to a sequence of goal elements (//object[@type='Goal']) from the document assigned to the variable $prometheusDoc.

    - let $prometheusDoc := doc(pdt:getPDTFileName())
    - let $prometheusGoals := $prometheusDoc//object[@type='Goal']

Sub-part 2.2 (ITERATION): It uses a *FLWOR* (FOR, LET, WHERE, ORDER BY, RETURN) expression in XQuery. The expression is composed of three parts. The first part selects elements defined in the variables (binding elements), the second part defines conditions to the rule, and the third part specifies actions to be executed. The actions can be concerned with the creation of traceability relation or the identification of missing elements in the models.

- Binding element – it consist of binding elements of a sequence to variables. For instance, the following statement binds elements of sequence in $SRTasks to the $SRTask variable and elements of the sequence in $prometheusGoal to $prometheusGoal variable.

    - for $SRTask in $SRTasks, $prometheusGoal in $prometheusGoals

- Condition – it defines the *condition* part of the rule that should be satisfied. Conditions can be defined by the where expression clause of a *for* clause in XQuery or by the test expression part of an if-then-else expression in XQuery. The condition part of the rule uses XQuery built-in functions and expressions, and the Java extra functions that we have developed. These extras functions contribute to the generation of the traceability relations and the identification of missing elements (see Section 4.2).

- Action: Traceability Relation Generation – it consist of returning an XML element to the Traceability_Completeness_Checking_Engine component that includes information about the traceability relation created. It contains the type of relation (*type*), an identifier of the rule that created the relation (*ruleID*), a percentage that indicates the level of confidence in the relation created (*degreeOfCompleteness*) and the information about the source and target element related. The source and target elements contain the location of the document where the element is stored (*doc*), type (*type*), name (*name*), and id (*id*) of the element. The degreeOfCompleteness represents the percentage of the condition of the rule that has been satisfied. For instance, in Figure 4.2 an *overlaps* traceability relation has been created between Organize Delivery SR task and Arrange Delivery SR task. This overlaps traceability relation has a degreeOfCompleteness of 66.7% that represents that only 66.7% of the condition of the rule has been satisfied. Although the traceability relation has been created, some elements are missing, or there are some discrepancies of names between sub-elements of the Organize Delivery SR task and sub-elements of Arrange delivery goal.

- Action: Missing Element Information Identification – it consist of returning an XML element to the Traceability_Completeness_Checking_Engine component describing information about missing elements in the models being compared. It contains information about the type (*typeSource*), name (*nameSource*), id (*idSource*), and document (*docSource*) of the element to which a representation is missing in the target document, and information about the type (*typeTarget*) and document (*docTarget*) of the missing element.

```
<Rule id="ruleID" priority="priorityNumber" type="relationType"
        elementTypeA="typeSourceElement" elementTypeB="typeTargetElement"
        description="descriptionText">
   <XQuery>
      <![CDATA[
    //DECLARE
         // Namespace declarations
            declare namespace name = "java:className";

         // Variable declarations
            let $variableName := XPathExpression

     // ITERATION
          //Binding elements
           for $elem_1 in $seq_1,
                $elem_2 in $seq_2,
                $elem_n in $seq_n
          // Condition
                 f_1(f_{1+1}…(f_{1+j}(·))…)
          // Action
            // Traceability Link Generation
              <TraceabilityRelation   type="relationType" ruleID="ruleID"
                      degreeOfCompleteness="similarityMeasure">
                      <Element doc="path" type="elementType" name="elementName"
                                   id="elementID">
                      </Element>
                     <Element…></Element>

            // Missing Element Information Generation
             <MissingElement typeSource="typeSourceElement"
                                   idSource="sourceElementID"
                                   nameSource="sourceElementName"
                                   docSource="documentPath"
                                   typeTarget="typeTargetElement"
                                   docTarget="documentPath">
             </MissingElement>
]]>
```

Figure 4.3 Rule Template

Our traceability rule template supports the definition of three types of traceability rules, namely:

Type 1: rules to create traceability relations and identify missing elements;

Type 2: rules to create traceability relations, and

Type 3: rules to identify missing elements.

We present below examples of these types of traceability rules

Type 1:

Figure 4.4 presents an example of a rule (Rule 4) to create traceability relations and identify missing elements between SR task in *i** and goals in Prometheus. In order to facilitate explanation, we divide the rule in Figure 4.4 in several parts.

```
<Rule id="Rule4"
        priority="1"
        type="overlaps"
        elementTypeA="SR Task"
        elementTypeB="Prometheus Goal"
        description="Rule identify traceability relations between SR Tasks in i* and Goals in
        Prometheus">
   <XQuery>
     <![CDATA[
          declare namespace f = "java:retratos.XQueryFunctions";
          declare namespace syn = "java:retratos.XQuerySynonymsFunctions";
          declare namespace sim = "java:retratos.XQuerySimilarityFunctions";
          declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
          declare namespace pdt = "java:retratos.XQueryPDTFunctions";
          declare namespace taom = "java:retratos.XQueryTAOMFunctions";
          declare namespace xmi="http://www.omg.org/XMI";
          let $istarDoc := doc(taom:getTAOMFileName())
          let $prometheusDoc := doc(pdt:getPDTFileName())
          let $systemActor :=   $istarDoc//TroposClasses[@xsi:type=
                                'it.itc.sra.taom4e.model.core.informalcore.formalcore:FActor'  and
                                                                                    @isSystem='true']
          let $SRTasks :=  $istarDoc//TroposClasses[@xsi:type='
                           it.itc.sra.taom4e.model.core.informalcore.formalcore:FPlan'  and
                           @Actor=$systemActor/@xmi:id]
          let $prometheusGoals := $prometheusDoc//object[@type='Goal']
          for $SRTask in $SRTasks, $prometheusGoal in $prometheusGoals
            return
              if ( f:clr() and syn:isSynonyms($SRTask/@name,
                           $prometheusGoal/base/field[@name='name']/text())  and
                           sim:isPositiveSimilar(pdt:getPrometheusSubElements($prometheusGoal,
                           "subGoals"), taom:getSubGoalsAndTask($SRTask),40.0))
             then
              <TraceabilityRelation type="overlaps" ruleID="rule4a"
                                    degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
                <Element doc="{taom:getTAOMFileName()}" name="{$SRTask/@name}"
                         type="SR Task" id="{$SRTask/@xmi:id}">
                </Element>
                <Element doc="{pdt:getPDTFileName()}" type="Goal"
                         name="{$prometheusGoal/base/field[@name='name']/text()}"
                         id="{$prometheusGoal/@id
                         {
                           for $i in (0 to cc:getNumberOfMissingElements())
                             return
                               <MissingElement typeSource="SR Task"
                                 idSource="{cc:getIDMissingElement($i)}"
                                 nameSource="{cc:getNameMissingElement($i)}"
                         name="{$prometheusGoal/base/field[@name='name']/text()}"
                         id="{$prometheusGoal/@id
                         {
                          for $i in (0 to cc:getNumberOfMissingElements())
                            return
                              <MissingElement typeSource="SR Task"
                                idSource="{cc:getIDMissingElement($i)}"
                                nameSource="{cc:getNameMissingElement($i)}"
                                docSource="{taom:getTAOMFileName()}"
                                typeTarget="Goal"
                               docTarget="{pdt:getPDTFileName()}">
                              </MissingElement>
                         }
          </Element>
        </TraceabilityRelation>
                        else
                         if ( f:clr() and
                             sim:isSimilar(pdt:getPrometheusSubElements($prometheusGoal,
                                     "subGoals"),taom:getSubGoalsAndTask($topLevelGoal),60.0))
                         then
                           <TraceabilityRelation type="overlaps" ruleID="rule1b"
                                     degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
                            <Element doc="{taom:getTAOMFileName()}"
                                      name="{$topLevelGoal/@name}"
                                      type="SD Goal"
                                      id="{$topLevelGoal/@xmi:id}">
                            </Element>
                            <Element doc="{pdt:getPDTFileName()}" type="Goal"
                                      name="{$prometheusGoal/base/field[@name='name']/text()}"
                                       id="{$prometheusGoal/@id}">
                                      {
                                       for $i in (0 to cc:getNumberOfMissingElements())
                                          return
                                           <MissingElement
                                                typeSource="SR Task"
                                                idSource="{cc:getIDMissingElement($i)}"
                                                nameSource="{cc:getNameMissingElement($i)}"
                                                docSource="{cc:getDocSourceMissingElement($i)}"
                                                typeTarget="{cc:getTypeTargetMissingElement($i)}"
                                                docTarget="{pdt:getPDTFileName()}">
                                            </MissingElement>
                                      }
                            </Element>
                          </TraceabilityRelation>
                         else
                           ""
    ]]>
  </XQuery>
</Rule>
```

Figure 4.4 Rule4

Figure 4.5 shows the main element of the rule (i.e. <Rule>) and its properties such as (i) unique identifier (i.e. Rule4), (ii) priority of execution of the rule (i.e. “1”), (iii) type of the relation created by the rule (i.e. “overlaps”), (iv) source element (i.e. “SR Task”), (v) target element (i.e. “Prometheus Goal”), and (vi) description of the rule (i.e. “Rule identify traceability relations between SR Tasks in *i** and Goals in Prometheus”).

```
<Rule id="Rule4"
        priority="1"
        type="overlaps"
        elementTypeA="SR Task"
        elementTypeB="Prometheus Goal"
        description="Rule identify traceability relations between SR Tasks in i* and Goals in
        Prometheus">
   <XQuery>
     <![CDATA[ …
     ]]>
   </XQuery>
</Rule>
```

Figure 4.5 Rule4 Header

Figure 4.6 shows the namespace declaration part of the rule. Each rule contains an XQuery element (i.e. <XQuery>) with the XQuery code. To avoid the text inside the XQuery element to be parsed by the XML parser, the XQuery is enclosed with a CDATA section. The declaration part defines namespaces used by the rule. For instance, the namespace *f, syn, sim, cc, pdt,* and *taom* allows the XQuery code to access functions implemented in XQueryFunctions, XQuerySynonymsFunctions, XQuerySimilarityFunctions, XQueryCompletenessCheckingFunctions, XQueryPDTFunctions, and XQueryTAOMFunctions classes, respectively. Section 4.2 provides a complete list with all the extended functions implemented in Java. Example of a function that is implemented in XQuerySynonymsFunctions class and is used in the Rule4, is *isSynonyms* function. A special case is the xmi namespace that is used to allow XPath expressions access elements defined in the *i** model created by the TAOME4E tool.

```
<XQuery> <![CDATA[
declare namespace f = "java:retratos.XQueryFunctions";
declare namespace syn = "java:retratos.XQuerySynonymsFunctions";
declare namespace sim = "java:retratos.XQuerySimilarityFunctions";
declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
declare namespace pdt = "java:retratos.XQueryPDTFunctions";
declare namespace taom = "java:retratos.XQueryTAOMFunctions";
declare namespace xmi="http://www.omg.org/XMI";
```

Figure 4.6 Namespace declarations

Figure 4.7 shows variable declarations for the Rule4. The first two statements assign to variables $*istarDoc* and $*prometheusDoc* the location of the *i** model and Prometheus model, respectively. The third statement assigns to $*systemActor* variable a sequence of actors in *i** that represent software systems (i.e. *isSystem* property equals 'true'). The fourth statement assigns to $SRTasks variable a sequence of SR tasks in *i** of an actor that represent a software system. The fifth statement assigns to the $prometheusGoal variable a sequence goal elements from the Prometheus model.

```
let $istarDoc := doc(taom:getTAOMFileName())
let $prometheusDoc := doc(pdt:getPDTFileName())
let $systemActor :=
   $istarDoc//TroposClasses[@xsi:type='it.itc.sra.taom4e.model.core.informalcore.formalcore:FActor'
                                                                          and @isSystem='true']
let $SRTasks :=
   $istarDoc//TroposClasses[@xsi:type='it.itc.sra.taom4e.model.core.informalcore.formalcore:FPlan'
                                                          and @Actor=$systemActor/@xmi:id]
let $prometheusGoals := $prometheusDoc//object[@type='Goal']
```

Figure 4.7 Variable declarations

Figure 4.8 shows the iteration part of Rule4. The *for* clause generates a sequence of tuples between SR tasks in *i** ($*SRTasks*) and goals in Prometheus ($*prometheusGoals*). The condition part of the rule is defined using *if-then-else* clauses.

The first condition (first *if* clause) checks if the name of a SR task in the tuple is synonyms (*isSynonyms* function) with the name of the goal in Prometheus, and if the sub-elements of the goals in Prometheus are similar (*isPositiveSimilar* function) to the sub-elements of SR tasks in *i**. The *isPositiveSimilar* function receives three parameters, namely (a) a list of sub-goals of a goal in Prometheus, (b) a list of goals and sub-tasks of a SR task in *i**, and (c) a threshold value (i.e. 40.0) that defines the degree of similarity that has to be achieved if the function returns true value. The second condition (second *if* clause in the *else* part of the first if clause) uses the *isSimilar* function to compare the similarity between sub-elements of a Prometheus goal and sub-tasks and sub-goals of a SR task in *i**. The difference between the *isPositiveSimilar* function and the *isSimilar* function is that the *isPositivieSimilar* function returns true if any of the list of sub-elements is empty. For instance, if the goal in Prometheus does not have sub-goals then *isPositiveSimilar* function always returns true. The reason why the threshold used by the second condition (i.e. 60.0) in the *isSimilar* function is greater than the threshold used by the first condition is because the condition is only based on the sub-elements. Therefore, we assume that the number of sub-elements that have to be similar in the second case (when name of a SR task in the tuple is not synonyms to the name of the goal in Prometheus) has to be greater than in the case of the first condition.

```
for $SRTask in $SRTasks, $prometheusGoal in $prometheusGoals
   return
   if ( f:clr() and
        syn:isSynonyms($SRTask/@name,$prometheusGoal/base/field[@name='name']/text()) and
        sim:isPositiveSimilar(pdt:getPrometheusSubElements($prometheusGoal,"subGoals"),
                                                          taom:getSubGoalsAndTask($SRTask),40.0))
   then
     …
   else
      if ( f:clr() and
           sim:isSimilar(pdt:getPrometheusSubElements($prometheusGoal,"subGoals"),
                                                          taom:getSubGoalsAndTask($SRTask),60.0))
      then
        …
      else
        ""
```

Figure 4.8 Condition part

Figure 4.9 shows the results of the traceability relation created by Rule4. The rule creates a *TraceabilityRelation* element when the condition of the rule holds. It returns an element in XML that is added to the Traceability_Relations_Missing_Elements file. The *TraceabilityRelation* element contains the type of relation that is created by the rule (i.e. “overlaps”), the identifier of the rule (i.e. “rule4a”), and the degree of completeness between the source and target elements that satisfies the rule condition.

In Figure 4.9, the degree of completeness is obtained from the *getDegreeOfCompleteness* function and it is calculated based on the similarities of the sub-elements of a SR task and a Prometheus goal. A traceability relation contains two elements related to the source and target elements that are related. In Figure 4.10, the first element refers to the SR task and contains as attributes (a) the name of the document that has the SR task that it is obtained from the *getTAOMFileName* function; (b) the name of the element that is obtained from the XPath expression *($SRTask/@name)*; (c) the type of the element that is “SR task”; and (d) the id of the element that is retrieved from the XPath expression (*$SRTask/@xmi:id*). The second element refers to the Prometheus goal and has similar properties: (a) the name of the document that contains the Prometheus goal and it is obtained from the *getPDTFileName* function; (b) the name of the element that is obtained from the XPath expression (*$prometheusGoal/base/field[@name='name']/text()*); (c) the type of the element that is "Prometheus Goal"; and (d) the id of the element that is retrieved from the XPath expression (*$prometheusGoal/@id*).

```
<TraceabilityRelation
   type="overlaps"
   ruleID="rule4a"
   degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
      <Element
         doc="{taom:getTAOMFileName()}"
         name="{$SRTask/@name}"
         type="SR Task"
         id="{$SRTask/@xmi:id}">
     </Element>
     <Element doc="{pdt:getPDTFileName()}"
                  type="Prometheus Goal"
                  name="{$prometheusGoal/base/field[@name='name']/text()}"
                  id="{$prometheusGoal/@id}">
                        …
     </Element>
  </TraceabilityRelation>
```

Figure 4.9 Traceability Relation Creation

Figure 4.10 shows an overlaps traceability relation created by the rule4a between Organize Delivery SR task and Arrange delivery goal in Prometheus with a degree of completeness of 66.7%. The degree of completeness is obtained from the *getDegreeOfCompleteness* function and it is calculated based on the similarities of the sub-elements of Organize Delivery SR task and Arrange delivery goal in Prometheus. Arrange delivery has three sub-goals (i.e., *Get delivery options*, *Log outgoing delivery* and *Calculate delivery*), while *Organise Delivery* SR task has three sub-tasks (i.e., *Obtain Delivery Options* and *Compute Delivery Time Estimates*, and *Place Delivery Request*). *Get delivery options* is synonyms to *Obtain Delivery Options* and *Calculate delivery time estimates* is synonyms to *Compute Delivery Time Estimates*. A sub-task concerned with sub-goal Log outgoing delivery is missing and the degree of completeness in this case is 2/3 = 66.7% (two out of three sub-tasks are similar to sub-goals).

```
<TraceabilityRelation ruleID="rule4a"  type="overlaps" degreeOfCompleteness="66,7">
   <Element doc="c:/ElectronicBookStore.pd" type="Goal" name="Arrange delivery" id="104"/>
   <Element  doc="c:/ElectronicBookshop.tropos"  type="SR Task" name="Organize Delivery"
          id="_ZYFmBVnqEduALZ_6XdllYA">
      <MissingElement>...</Missing Element>
  </Element>
  <Element doc="c:/ElectronicBookStore.pd" type="Goal" name="Arrange delivery" id="104"/>
</TraceabilitvRelation>
```

Figure 4.10 Traceability Relation between Arrange delivery and Organize delivery

Figure 4.11 shows the results of the identified missing elements created by Rule4. The Missing Element contains the following attributes: (a) the type of the source element to which a representation is missing from the target document (i.e. SR Task); (b) the id of the source element to which a representation is missing from the target document that is retrieved from *getIDMissingElement*($i) function; (c)the name of the element to which a representation is missing from the target document that is obtained from *getNameMissingElement($i)* function; (d) the name of the document from the source element that is retrieved by *getTAOMFileName* function; (e) type of the target element (i.e. "Prometheus Goal"); and (f) name of the target document received from the *getPDTFileName*() function.

```
<MissingElement
     typeSource="SR Task"
     idSource="{cc:getIDMissingElement($i)}"
     nameSource="{cc:getNameMissingElement($i)}"
     docSource="{taom:getTAOMFileName()}"
     typeTarget="Prometheus Goal"
     docTarget="{pdt:getPDTFileName()}">
</MissingElement>
```

Figure 4.11 Generation of Missing Element

Figure 4.12 shows an example of the *LogOutgoingDelivery* goal in Prometheus that is missing in the *i** model. In this case there are two possibilities to restore the models for the missing element. The first possibility is to create a sub-task or sub-goal of the Organize Delivery SR task named "LogOutgoingDelivery" in the *i** model. The second possibility is to change the name given to a sub-task or sub-goal of the Organize Delivery SR task that should have a name that is synonym to "LogOutgoingDelivery". Based on this information and analysing the model (see Figure 4.2) the user can conclude that one sub-task or sub-goal named "LogOutgoingDelivery" is missing and should be created in the *i** model.

```
<TraceabilityRelation  ruleID="rule4a"  type="overlaps" degreeOfCompleteness="66,7">
…
      <MissingElement  docSource=="c:/users/by916/ElectronicBookStore.pd"
                       typeSource="Goal"
                       nameSource=" Log Outgoing Delivery"
                       idSource="98 "
                       typeTarget="SR Task or SR Goal"/>
  …
</TraceabilityRelation>
```

Figure 4.12 Log Outgoing Delivery Missing Element

Type 2:

Figure 4.13 shows an example of a rule (Rule49) that identifies overlaps traceability relations between actors in *i** and agent in Prometheus. Here, we do not describe in details the declarations of namespaces and variables since they are similar to these declarations in Rule4 (see Figure 4.4). As shown in Figure 4.13, the main difference for the declaration of variables between Rule49 and Rule4 are variable $*systemActors,* which contains all actors in *i** and $*prometheusAgents* that contains all agents in Prometheus.

```
<Rule id="Rule49" priority="1" type="overlaps" elementTypeA="Istar Actor"
    elementTypeB="Prometheus Agent"
    description="Rule identify traceability relations between Actors in i* and Agents in Prometheus">
    <XQuery>
      <![CDATA[
        declare namespace f = "java:retratos.XQueryFunctions";
        declare namespace syn = "java:retratos.XQuerySynonymsFunctions";
        declare namespace sim = "java:retratos.XQuerySimilarityFunctions";
        declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
        declare namespace pdt = "java:retratos.XQueryPDTFunctions";
        declare namespace taom = "java:retratos.XQueryTAOMFunctions";
        declare namespace xmi="http://www.omg.org/XMI";

         let $istarDoc := doc(taom:getTAOMFileName())
         let $systemActors := $istarDoc//TroposClasses[@xsi:type=
                              'it.itc.sra.taom4e.model.core.informalcore.formalcore:FActor'
                              and @isSystem='true']
         let $prometheusDoc := doc(pdt:getPDTFileName())
         let $prometheusAgents := $prometheusDoc//object[@type='Agent']
         for $systemActor in $systemActors, $prometheusAgent in $prometheusAgents
             return
               if ( f:clr() and
                     syn:isSynonyms($systemActor/@name,
                              $prometheusAgent/base/field[@name='name']/text()) )
               then
                 <TraceabilityRelation type="overlaps"  ruleID="rule49a"
                                  degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
                   <Element doc="{taom:getTAOMFileName()}"
                              name="{$systemActor/@name}"
                              type="Actor" id="{$systemActor/@xmi:id}">
                   </Element>
                   <Element doc="{pdt:getPDTFileName()}" type="Agent"
                              name="{$prometheusAgent/base/field[@name='name']/text()}"
                              id="{$prometheusAgent/@id}">
                    </Element>
                 </TraceabilityRelation>
               else
                 ""
        ]]>
  </XQuery>
</Rule>
```

Figure 4.13 Rule49

Figure 4.14 shows the iteration part of the Rule49. The *for* clause generates a sequence of tuples between actors in *i** (*$systemActors*) and agents in Prometheus (*$prometheusAgent*). The condition part of the rule is defined using *if-then-else* clause. The condition checks if the name of an actor in the tuple is synonyms (*isSynonyms* function) to the name of the agent in the tuple. The action part of the rule, creates a *TraceabilityRelation* when the condition is satisfied (i.e., when the name of the actor is synonyms to the name of the agent). The content of element Element is similar to the content described in the Rule4 for element Element.

In order to illustrate Rule49, consider Airport agent in Prometheus and Airport actor in *i** (see Figure 4.15). Since the Airport actor has a synonyms name to Airport agent, an overlaps traceability relation is created between Airport actor in *i** and Airport agent in Prometheus (see Figure 4.16).

```
for $systemActor in $systemActors, $prometheusAgent in $prometheusAgents
            return
              if ( f:clr() and
                    syn:isSynonyms($systemActor/@name,
                              $prometheusAgent/base/field[@name='name']/text()) )
              then
                <TraceabilityRelation type="overlaps"  ruleID="rule49a"
                                    degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
                  <Element doc="{taom:getTAOMFileName()}"
                            name="{$systemActor/@name}"
                            type="Actor" id="{$systemActor/@xmi:id}">
                  </Element>
                  <Element doc="{pdt:getPDTFileName()}" type="Agent"
                            name="{$prometheusAgent/base/field[@name='name']/text()}"
                            id="{$prometheusAgent/@id}">
                  </Element>
                </TraceabilityRelation>
              else
                ""
```

Figure 4.14 Iteration part of the Rule15

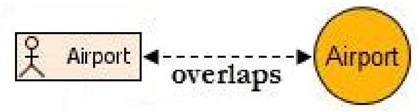


Figure 4.15 Airport agent in Prometheus and Airport actor in *i**

```
<TraceabilityRelation ruleID="rule49a" degreeOfCompleteness="100" type="overlaps">
  <Element id="_9GxkMFyvEd6qIOGYcZQlag" name="Airport"
        doc="file:///C:/ElectronicBookstore2.tropos" type="Actor"/>
  <Element id="132" name="Airport" doc="file:///C:/AirTrafficControl2.pd" type="Agent"/>
</TraceabilityRelation>
```

Figure 4.16 Traceability relation between Airport agent and Airport actor

Type 3:

Figure 4.17 shows an example of a rule (Rule 4cc) that identifies missing elements (completeness checking) between SR tasks of an actor in *i** model and Prometheus model. Here, we do not describe in details the declarations of namespaces and variables since they are similar to these declarations in Rule4 (see Figure 4.5). As shown in Figure 4.17, the main difference for the declaration of variables between Rule4 and Rule4cc, are variable $*overlapsLink,* which contains a sequence of *overlaps* relations between all SR tasks in *i** and elements in the Prometheus model, and variable $*SRPlans*, which contains all SR tasks in *i** model.

```
<Rule id="Rule4cc"  priority="2" type="overlaps"
    elementTypeA="SR Plan"
    elementTypeB="Prometheus Goal"
   description="Check if every SR Task that a system actor has to achieve is represented as goal in
Prometheus">
    <XQuery>
      <![CDATA[
        declare namespace xmi="http://www.omg.org/XMI";
        declare namespace f = "java:retratos.XQueryFunctions";
        declare namespace pdt = "java:retratos.XQueryPDTFunctions";
        declare namespace taom = "java:retratos.XQueryTAOMFunctions";

         let $istarDoc := doc(taom:getTAOMFileName())
         let $systemActor :=  $istarDoc//TroposClasses[@xsi:type=
                              'it.itc.sra.taom4e.model.core.informalcore.formalcore:FActor'
                                                                    and @isSystem='true']
         let $SRPlans := istarDoc//TroposClasses[@xsi:type=
                       'it.itc.sra.taom4e.model.core.informalcore.formalcore:FPlan'
                                                     and @Actor=$systemActor/@xmi:id]
         let $traceabilityDoc := doc(f:getTraceabilityFileName())
         let $overlapsLink := $traceabilityDoc//TraceabilityRelation[@type=
                                                 'overlaps']/Element[@type='SR Task']
                   …
       ]]> </XQuery></Rule>
```

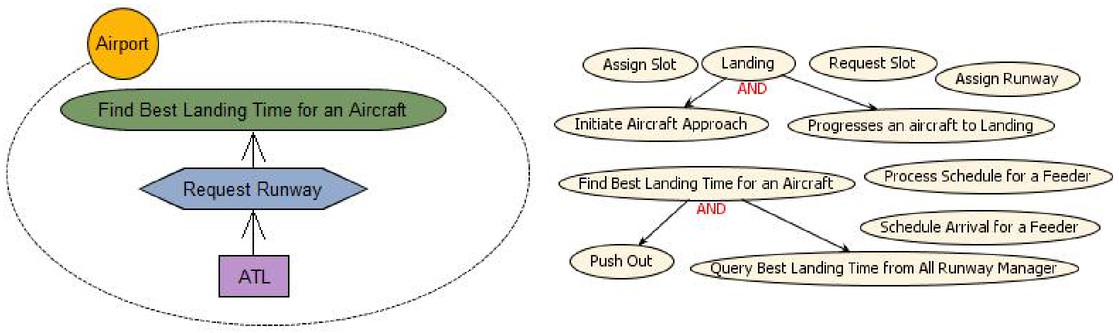
Figure 4.17 Rule4cc

Figure 4.18 shows the iteration part of the Rule4cc. The *for* clause iterates on all tasks in *SRPlans* variable. The condition part of the rule verifies if there are no *overlaps* traceability relations between SR tasks in *i** and an element in Prometheus model. The action part of the rule, creates a *MissingElement* when the condition is verified (i.e., when there are no overlaps relations between SR tasks and Prometheus elements). The content of element MissingElement is similar to the content described in the Rule4 for element MissingElement.

```
for $SRPlan in $SRPlans
    where (not(some $link in $overlapsLink satisfies $link/@id =$SRPlan/@xmi:id))
    return
      <TraceabilityRelation
          type="overlaps"
           ruleID="rule4cc"
           degreeOfCompleteness="0">
              <MissingElement
                  typeSource="SR Plan"
                  idSource="{$SRPlan/@id}"
                  docSource="{taom:getTAOMFileName()}"
                  typeTarget="Prometheus Goal "
                  docTarget="{pdt:getPDTFileName()}">
           </MissingElement>
        </TraceabilityRelation>
```

Figure 4.18 Iteration part of the Rule4cc

In order to illustrate Rule4CC, consider an extract of the *i** SR model and Prometheus goal model for the Air Traffic Control System[1], given in Figure 4.19. Figure 4.20 shows the results of executing Rule4cc for the Request Runway SR task in *i** and Prometheus goal for the models presented in Figure 4.19. In this case, there are two possibilities to rectify the models. One possibility is concerned with the creation of a goal in Prometheus named "Request Runway". The other possibility is concerned with discrepancy between the name given to a goal in Prometheus that should have an overlaps relation with the "Request Runway" SR task in *i**. After analysis of the Prometheus model (see Appendix C), it is necessary to create a goal in Prometheus named "Request Runway".



Airport
Find Best Landing Time for an Aircraft
Request Runway
ATL
Assign Slot
Landing
Request Slot
Assign Runway
AND
Initiate Aircraft Approach
Progresses an aircraft to Landing
Find Best Landing Time for an Aircraft
Process Schedule for a Feeder
AND
Schedule Arrival for a Feeder
Push Out
Query Best Landing Time from All Runway Manager

Figure 4.19 Airport SR model and ATCE Prometheus Goal

[1] A full description of these models are presented in Appendix D.

```
<TraceabilityRelation ruleID="rule4cc" degreeOfCompleteness="0" type="overlaps">
  <MissingElement docTarget="c:/AirTrafficControl1.pd"
              idSource="_Oj5iYFywEd6qIOGYcZQlag"
              nameSource="Request Runway"
              typeSource="SR Task"
              docSource="C:/ElectronicBookstore1.tropos"
              typeTarget="Prometheus Goal"/>
</TraceabilityRelation>
```

Figure 4.20 Request Runway goal missing in Prometheus

4.3 Extended Functions

As described in Subsection 4.1, our framework uses different functions that we have developed to support the rules. These functions are classified in seven classes of functions, as described below.

- XQueryCompletenessCheckingFunctions class – It contains a list of methods in Java that extends XQuery to perform completeness checking.

- XQueryFunctions class – It contains a list of methods in Java that extends XQuery with general functionalities such as returns the name of traceability file name (*getTraceabilityFileName* function*)* and returns the value of an attribute of an XML Element in Saxon (*getAttributeValue* function).

- XQueryJACKFunctions class - It provides a list of methods in Java that that extends XQuery with functions to manipulate elements in the JACK XML file.

- XQueryPDTFunctions class – It provides a list of methods in Java that extends XQuery with functions to manipulate elements created by the PDT tool version 3.2.

- XQuerySimilarityFunctions class – It includes a list of methods in Java that extends XQuery with functions to compare similarities between elements in the models.

- XQuerySynonymsFunctions class – It contains a list of methods in Java that extends XQuery with functions to verify if names of elements in the models are synonyms.

- XQueryTAOMFunctions class – It provides a list of methods in Java that extends XQuery with functions to manipulate elements in *i** models created using the TAOME4E tool.

To use an XQuery extended function implemented in a Java class, it is necessary to first declare the class name that includes the function and then call the function wanted, as shown in the examples in section 4.2. Figure 4.21 shows an example when the pdt namespace is associated to XQueryPDTFunction class in Java (*declare namespace pdt = java:retratos.XQueryPDTFunctions*). The *getPDTFileName* function is invoked using the namespace followed by colon and the function name (pdt:getPDTFileName( )).

```
declare namespace pdt = "java:retratos.XQueryPDTFunctions"; ….

let $pdtDoc := doc(pdt:getPDTFileName())
```

Figure 4.21 Calling getPDTFileName extended function in Java

In the next sub-section we describe the functions in the different classes of functions that we have developed to support the framework.

4.3.1 Completeness checking functions

XQueryCompletenessCheckingFunctions class extends XQuery with functions to perform completeness checking. The main function is *completenessChecking* that verify if a list of elements A contains elements that are synonyms to a list B. The *getDegreeOfCompleteness* function returns the degree of similarity between the two lists. For instance, if all elements in list A are synonyms to elements in list B then the function returns 1.00 and if no elements in list A is synonyms to list A then the function returns 0. The class also contains functions to returns the filename that contains elements in list A that are missing to be represented in list B (i.e. there is no synonyms in list B), to returns the number of elements, name, id, type of elements in list A

In order to illustrate, consider the completenessChecking function that verifies if two lists of elements are similar. The function receives two lists of elements and returns true if the names of the elements in the lists are synonyms and returns false otherwise. Figure 4.22 shows ListA with element names "Login outgoing delivery", "Calculate delivery time estimates", and "Get delivery options", and ListB with element names "Obtain Delivery Options", "Compute Delivery Estimates" and "Place Delivery Request". The completenessCheching function checks if each element of ListA has a synonyms in ListB. In the Figure 4.22, “Get delivery options” is synonym of “Obtain Delivery Options”, and “Calculate delivery time estimates” is synonym of “Compute Delivery Time Estimates”. Since “Login outgoing delivery” does not have a synonym element in List B.

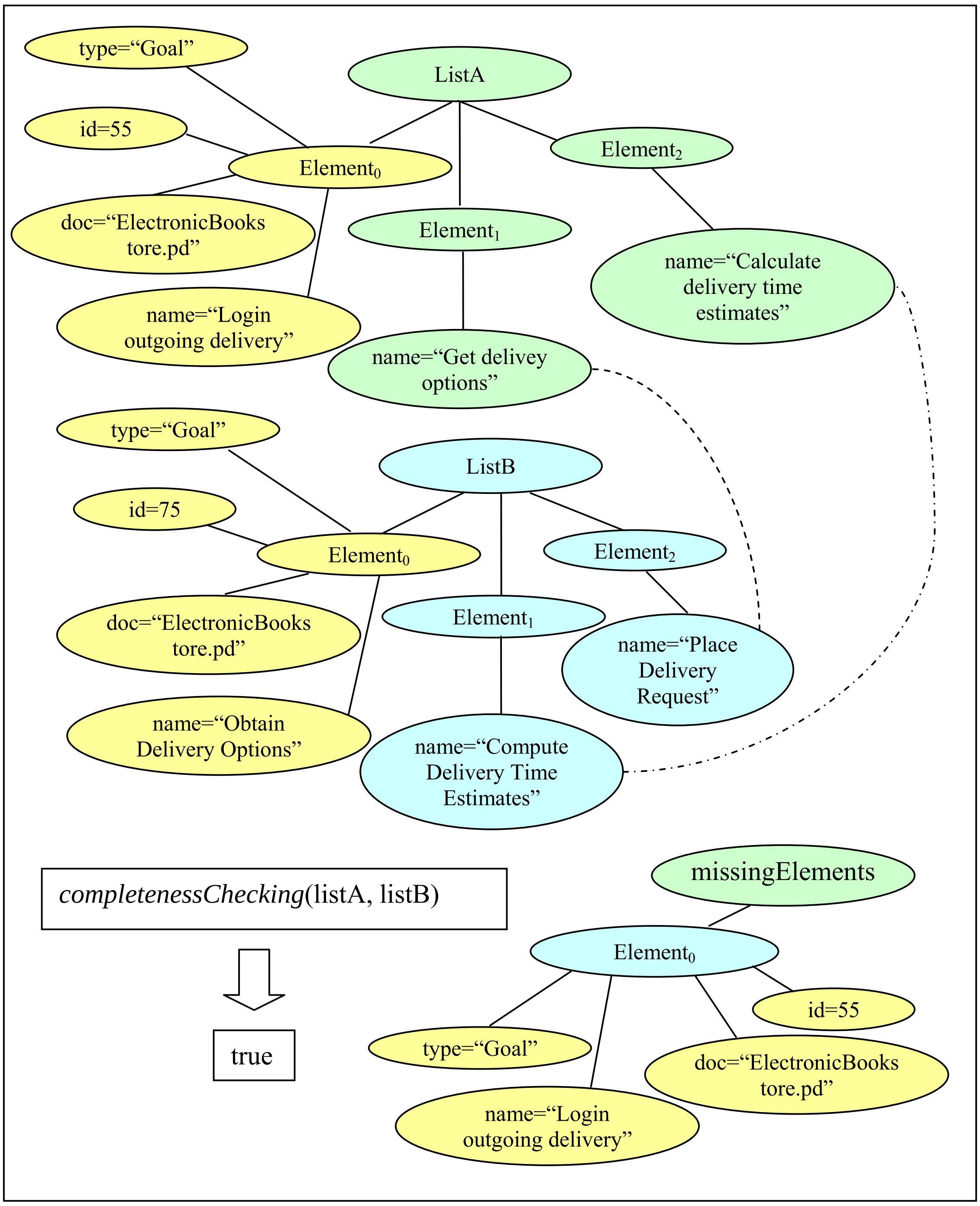

type=“Goal”
ListA
id=55
Element$_0$
Element$_2$
doc=“ElectronicBookstore.pd”
Element$_1$
name=“Calculate delivery time estimates”
name=“Login outgoing delivery”
name=“Get delivey options”
type=“Goal”
ListB
id=75
Element$_0$
Element$_2$
doc=“ElectronicBookstore.pd”
Element$_1$
name=“Place Delivery Request”
name=“Obtain Delivery Options”
name=“Compute Delivery Time Estimates”
completenessChecking(listA, listB)
missingElements
Element$_0$
true
id=55
type=“Goal”
doc=“ElectronicBookstore.pd”
name=“Login outgoing delivery”

Figure 4.22 List of strings

The "Login outgoing delivery" element is added to a list the missing elements (i.e. class variable of the XQueryFunctions class).

4.3.2 XQuery functions

XQueryFunctions class extends XQuery with general functionalities. XQueryFunctions class contains functions to check if a capability in Prometheus uses a SD Resource in *i**, to check if a capability in Prometheus uses a SR Resource in *i**, to initialize variables used by the created functions, to check if a list A of Strings contains all the elements of a list B of Strings, to check if a list A of Strings contains a specific String, to return the value of an attribute of an XML Element in Saxon, to return the name of filename of the file that contains the traceability relations, and to check if an element has an certain type of relation.

In order to illustrate, consider the capabilityUsesSDResource function. This function checks if a capability uses a SD resource in the *i** model. The function receives as a parameter a TinyNodeImpl element (i.e. XML Node in the Saxon tool) that represents a capability in XML and TinyNodeImpl element (i.e. Node in the Saxon tool) that represents a SD resource in XML and returns *true* if the capability includes a message that has an *overlaps* traceability relation with a SD resource. Figure 4.23 shows an example of ATL SD resource and Arrival Sequencing capability that includes Aircraft Event message. The ATL SD resource has an *overlaps* traceability relation with Aircraft Event message. If you call *capabilityUsesSDResource* function (see Figure 4.24) and pass as argument the Arrival Sequencing capability element in XML and the ATL SR Resource element in XML, the function first recovers all messages that the capability contains and then checks if there is some overlaps traceability relation between message and SD resource using the *isOverlap* function (see Section 4.2 to the description of the *isOverlap* function).

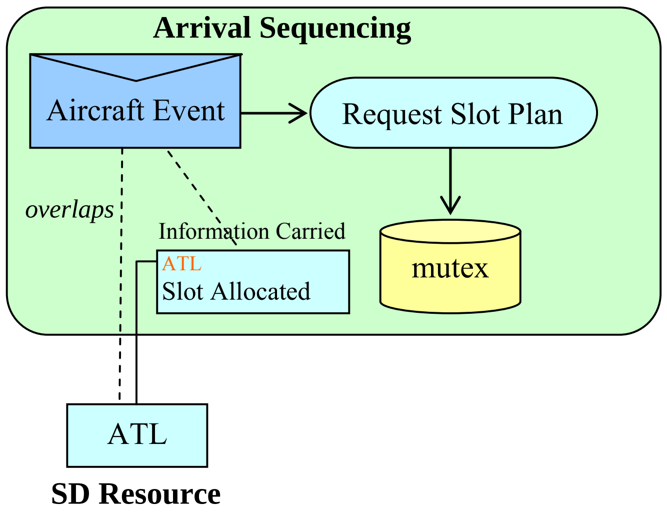
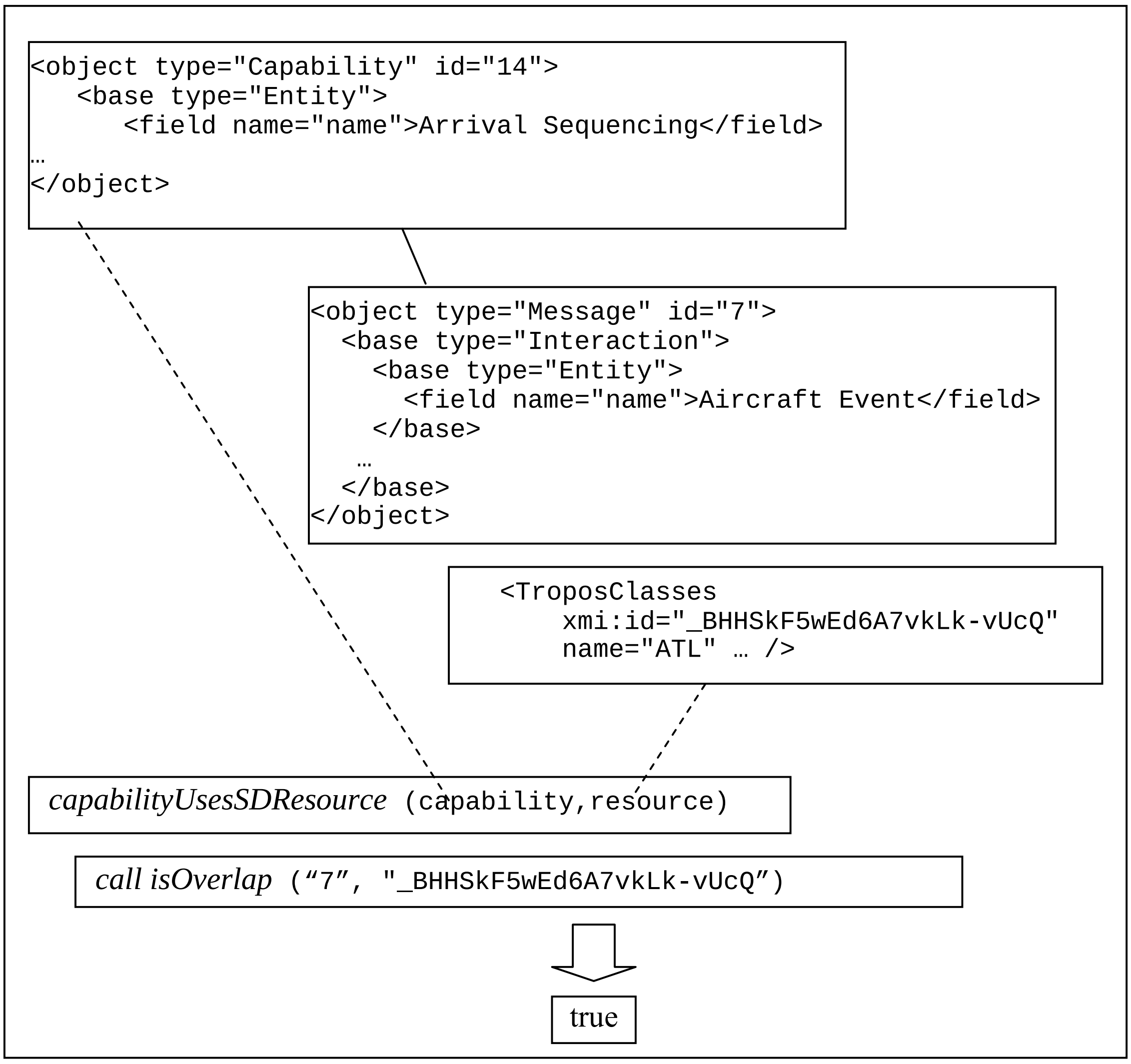


Figure 4.23 Arrival Sequencing Capability and ATL SD Resource



<object type="Capability" id="14">
<base type="Entity">
<field name="name">Arrival Sequencing</field>
…
</object>
<object type="Message" id="7">
<base type="Interaction">
<base type="Entity">
<field name="name">Aircraft Event</field>
</base>
…
</base>
</object>
<TroposClasses
xmi:id="_BHHSkF5wEd6A7vkLk-vUcQ"
name="ATL" … />
capabilityUsesSDResource (capability,resource)
call isOverlap ("7", "_BHHSkF5wEd6A7vkLk-vUcQ")
true

Figure 4.24 ***capabilityUsesSDResource function example***

4.3.3 XQueryJACKFunctions

XQueryJACKFunctions class extends XQuery with functions to manipulate elements in the JACK XML file. The XQueryJACKFunctions class contains a function that return the list of fields of a beliefSet in JACK and a function that returns the name of the file that contains the JACK code in XML.

In order to illustrate, consider the *getBeliefSetFields* function. This function receives as parameter the id of a beliefSet in Prometheus created using the PDT tool and returns a list of Field elements of this beliefSet. Figure 4.25 shows an example when the *getBeliefSetFields* function is called with id=58 (the id of landing_info data in Prometheus). The *getBeliefSetFields* function returns a list with the included fields.

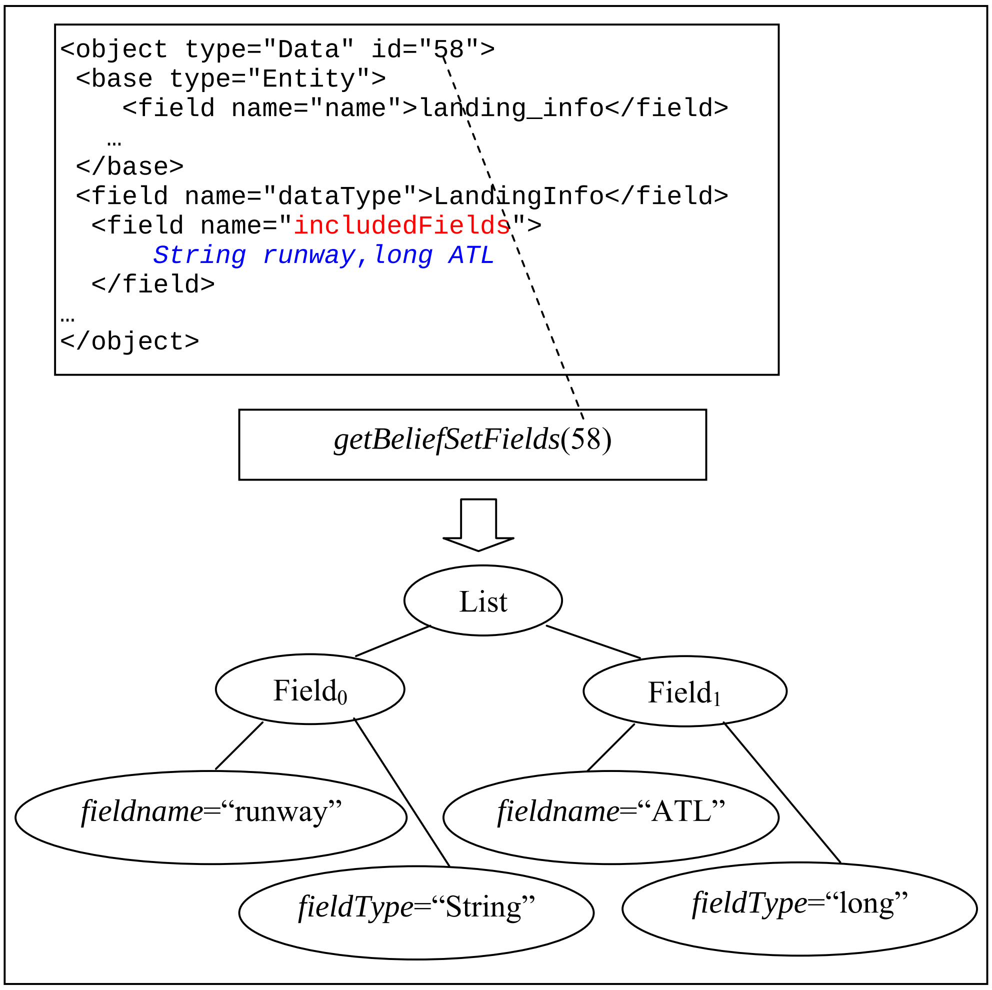
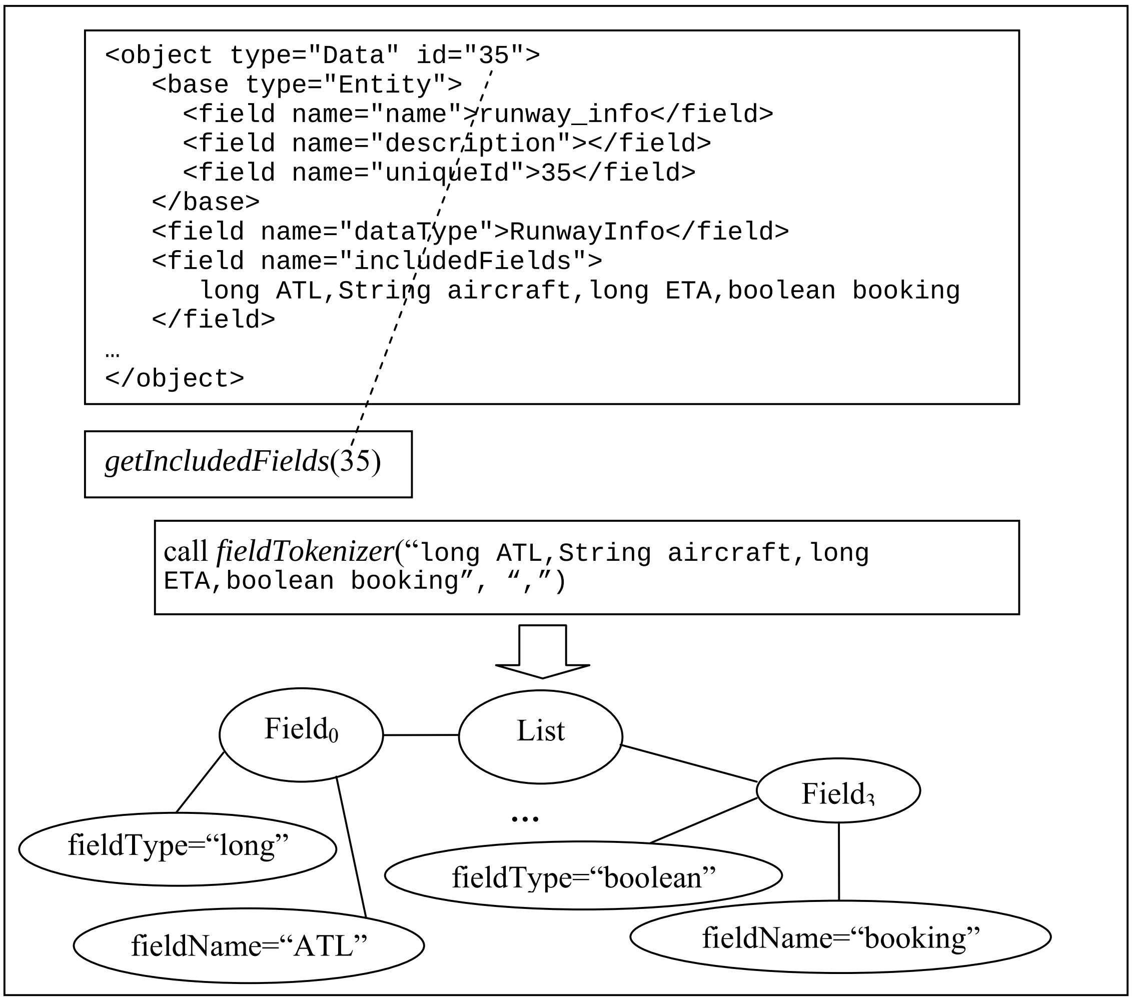

```
<object type="Data" id="58">
 <base type="Entity">
    <field name="name">landing_info</field>
   …
 </base>
 <field name="dataType">LandingInfo</field>
  <field name="includedFields">
      String runway,long ATL
  </field>
…
</object>
```

Figure 4.25 getBeliefSetFields function example

4.3.4 XQueryPDTFunctions

XQueryPDTFunctions class extends XQuery with functions to manipulate elements created by the PDT tool version 3.2. XQueryPDTFunctions contains functions to find all actors in *i** that uses a capability in Prometheus, to return included fields of a beliefSet in Prometheus, to return information carried of a percept in Prometheus, to return the filename of the file that contains the PDT file, to return sub-elements of an element in PDT, to return the steps of a scenario, to return sub-goals of a goal in PDT, return a list of the data used by an element in Prometheus, to verify if elements in Prometheus has relations (e.g. if a data is produced by a role in Prometheus).

In order to illustrate, consider *getIncludedFields* function. This function receives as parameter the id of an element in Prometheus and returns a list of *Fields* that contains information about the type and name of each included field. Figure 4.26 shows an example when the *getIncludedFields* function is called with id = 35 (the id of the *runway_info* data in Prometheus). The getIncludedFields function calls *fieldTokenizer* function passing as parameter a String containing included fields, and a String containing "," that is the delimiter used to separate the included fields. The *fieldTokenizer* function returns a list of *Fields* that contains the information about the include fields (e.g. *fieldType*="long", *fieldName*="ATL").



```
<object type="Data" id="35">
   <base type="Entity">
     <field name="name">runway_info</field>
     <field name="description"></field>
     <field name="uniqueId">35</field>
   </base>
   <field name="dataType">RunwayInfo</field>
   <field name="includedFields">
      long ATL,String aircraft,long ETA,boolean booking
   </field>
…
</object>
```

getIncludedFields(35)

call *fieldTokenizer*("long ATL,String aircraft,long ETA,boolean booking", ",")

Figure 4.26 getIncludesFields function example

4.3.5 XQuerySimilarityFunctions

The XQuerySimilarityFunctions class that extends XQuery with functions to compare similarities between elements in the models. The XQuerySimilarityFunction contains functions to verify if there is a creates traceability between two elements, verify if there is a creates traceability relation between two elements, to verify if there is an uses traceability relation between two elements, to verify if there is an overlaps traceability relation between two elements, to compare if two list of elements are similar, to count the number of elements in list that are similar to elements in the list B based on overlaps traceability relation, compare if two list of elements are similar based on overlaps traceability relations, to compare if a capability in JACK is similar to a capability in Prometheus, to compare if a data in Prometheus is similar to a beliefSet in JACK, to compare if a SD resource in *i** is similar to a message in Prometheus, to verify if an element has an overlaps traceability relation with any element in a list A.

In order to illustrate, consider the *isSimilar* function. This function receives two lists of elements and verifies if the number of elements in the *list1* that have names that are synonym to the names of elements in *list2,* is greater than a threshold. If the *list1* is empty then *isSimilar* function returns true. In order to illustrate, consider the example in Figure 4.27. The function compares if the name of each element in list1 has a synonym with elements in list2. The only element in list1 that does not have a synonym in list2 is "Login outgoing delivery". In this case, the percentage of elements in list1 that has a synonym in list2 is 66.7 that are greater than the threshold (i.e. 40).

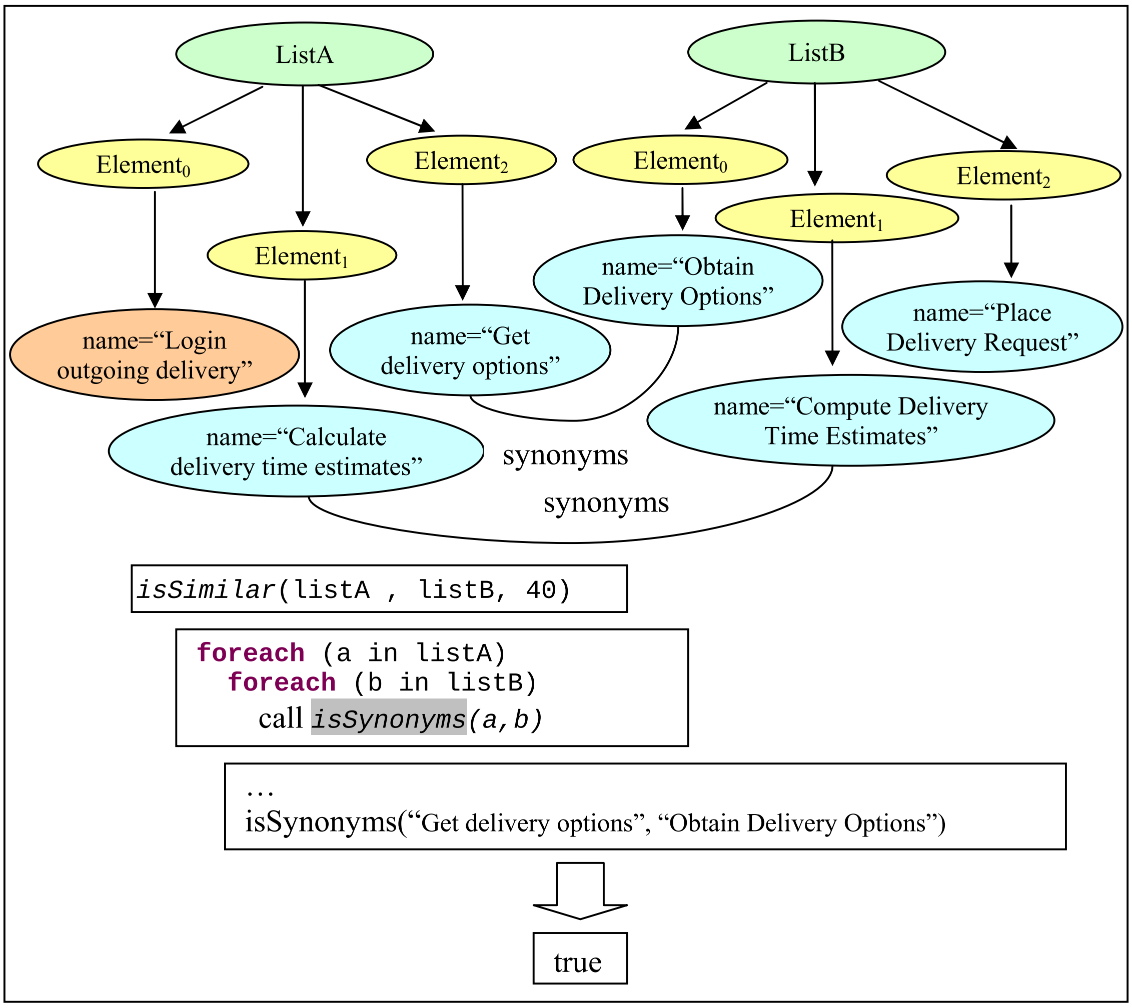

ListA
ListB
Element0
Element1
Element2
Element0
Element1
Element2
name="Login outgoing delivery"
name="Calculate delivery time estimates"
name="Get delivery options"
name="Obtain Delivery Options"
name="Compute Delivery Time Estimates"
name="Place Delivery Request"
synonyms
synonyms
isSimilar(listA , listB, 40)
foreach (a in listA)
foreach (b in listB)
call isSynonyms(a,b)
…
isSynonyms("Get delivery options", "Obtain Delivery Options")
true

Figure 4.27 isSimilar function example

4.3.6 XQuerySynonymsFunctions

The XQuerySynonymsFunctions class extends XQuery with functions to verify if names of elements in the models are synonyms. The XQuerySynonymsFunctiions class contains function to verify if a list A of strings contains another list B of strings based on synonyms, to verify if a list A of strings contains a string based on synonyms, to verify if two strings are synonyms, to break down a string into tokens.

In order to illustrate, consider *isSynonyms* function. This function receives as parameters *str1* and *str2* Strings. The *isSynonyms* function uses *stringTokenizerByUpperCase* function to break *str1* and *str2* in two lists of words, *wordList1* and *wordList2*. The *isSynonyms* function call *contains* function to each word in *wordList1* to verify if the word is contained in the *wordList2*. A word w is contained in the wordList2, if w is synonyms to a word in wordList2. Figure 4.28 shows an example when *isSynonyms* function receives as parameter "Get Delivery Options" and "Obtain Delivery Options" Strings. The *isSynonyms* function uses *stringTokenizerByUpperCase* function to break "Get Delivery Options" and "Obtain Delivery Options" in two lists of words, {"Get","Delivery","Options"} and {"Obtain", "Delivery", "Options"}. The *isSynonyms* function call *contains* function to Get, Delivery, and Options to verify if they are contained in {"Obtain", "Delivery", "Options"}. Figure 4.29 shows an example when *isSynonyms* function call *contains* function passing "Get" and {"Obtain", "Delivery", "Options"} as parameter. Get is synonyms to "Obtain", therefore the function returns true.

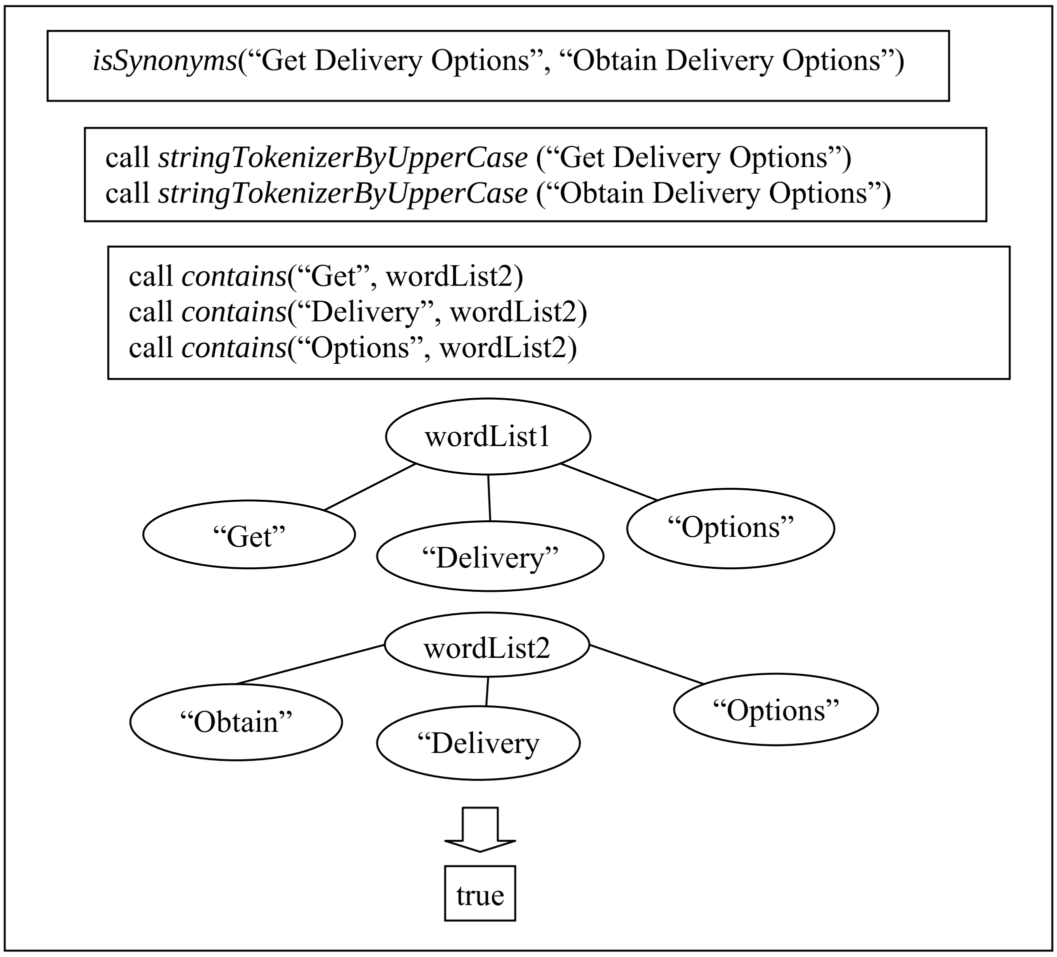

isSynonyms("Get Delivery Options", "Obtain Delivery Options")
call *stringTokenizerByUpperCase* ("Get Delivery Options")
call *stringTokenizerByUpperCase* ("Obtain Delivery Options")
call *contains*("Get", wordList2)
call *contains*("Delivery", wordList2)
call *contains*("Options", wordList2)
wordList1
"Get"
"Delivery"
"Options"
wordList2
"Obtain"
"Delivery
"Options"
true

Figure 4.28 isSynonyms function example

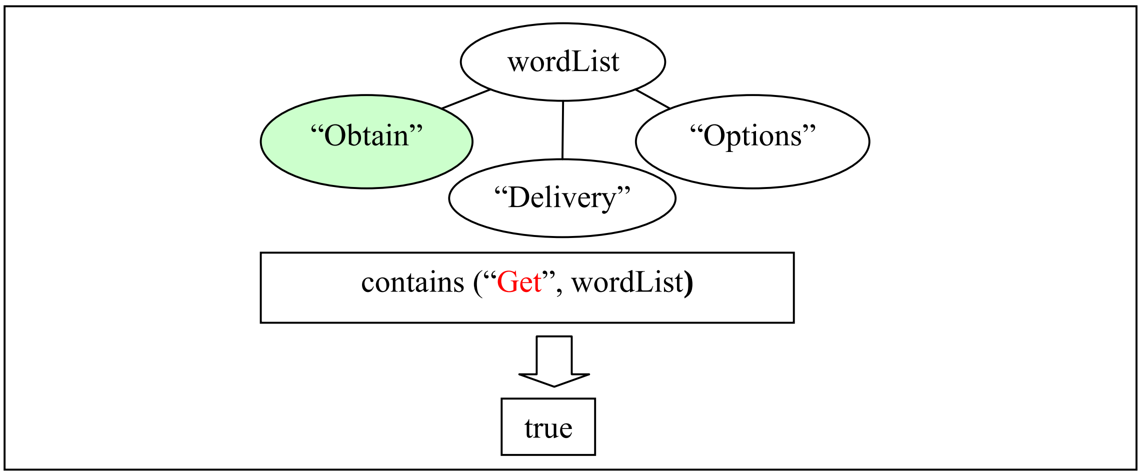
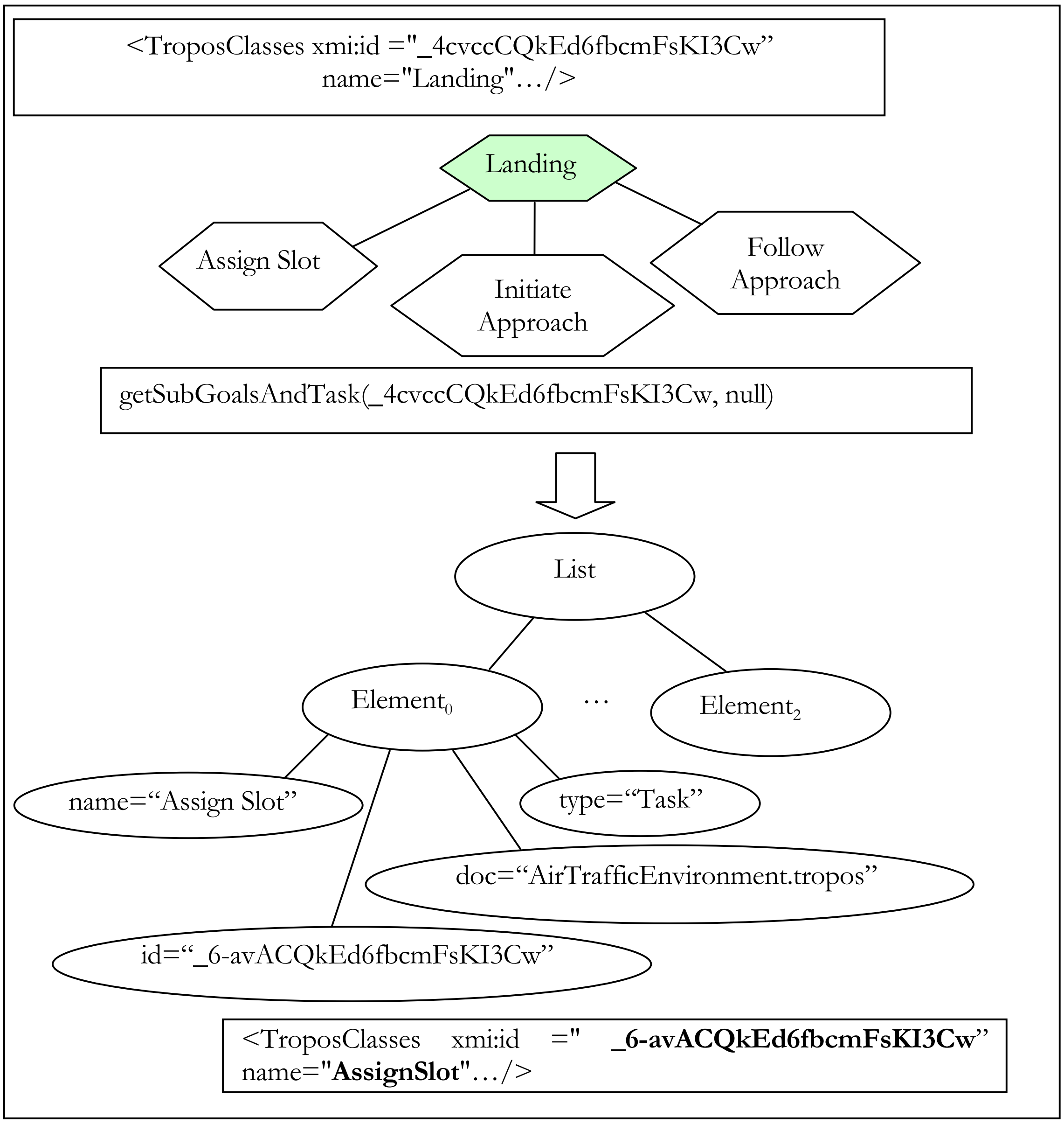


Figure 4.29 contains function example

4.3.7 XQueryTAOMFunctions

The XQueryTAOMFunctions class extends XQuery with functions to manipulate elements created by TAOM tool version 3.2. The XQueryTAOMFunctions class contains function to return the attribute value of an element in TAOME, to return a list of sub-elements of an element, to return a list of sub-goals and sub-tasks of an element and a function to return the filename of the file that contains the TAOME file.

In order to illustrate, consider the *getSubGoalsAndTask* function. This function receives an id of an element in *i** and returns the sub-goals and sub-tasks that are part of means-end and decomposition links with the *i** element. If a sub-element has sub-elements then the function calls itself recursively. Figure 4.30 shows an example when the function is called to retrieve sub-elements of the Landing task (xmi:id = "_4cvccCQkEd6fbcmFsKI3Cw") in *i**. The function returns a list of elements that consists of the Assign Slot, Initiate Approach and Follow Approach elements.



<TroposClasses xmi:id ="_4cvccCQkEd6fbcmFsKI3Cw" name="Landing".../>
Landing
Assign Slot
Initiate Approach
Follow Approach
getSubGoalsAndTask(_4cvccCQkEd6fbcmFsKI3Cw, null)
List
Element0
...
Element2
name="Assign Slot"
type="Task"
doc="AirTrafficEnvironment.tropos"
id="_6-avACQkEd6fbcmFsKI3Cw"
<TroposClasses xmi:id =" _6-avACQkEd6fbcmFsKI3Cw" name="AssignSlot".../>

Figure 4.30 getSubGoalsAndTask function example

4.4 Retratos Tool

In order to support our traceability framework, we have developed a prototype tool to identify missing elements and automatically generate traceability relations between *i** and Prometheus, and between Prometheus and JACK code, called Retratos. Retratos tool (see Figure 4.31) allows the user to create a new project and to define the location of models and rules that will be used during the generation of traceability relations and identification of missing elements. Retratos tool supports also functionalities to generate reports of the traceability relations and to visualise rules defined in the project, as well as an editor to create new rules. The rest of this section describes how to use Retratos tool.

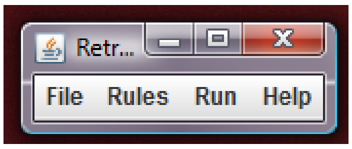


Figure 4.31 Retratos main menu

To start to use the tool the user has to select the ***File*** menu option followed (see Figure 4.32) by the menu item ***New Project***.

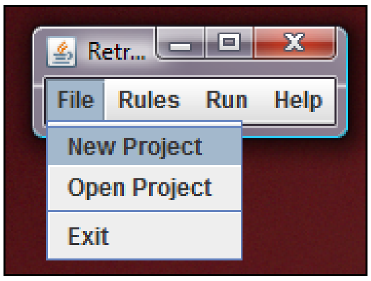
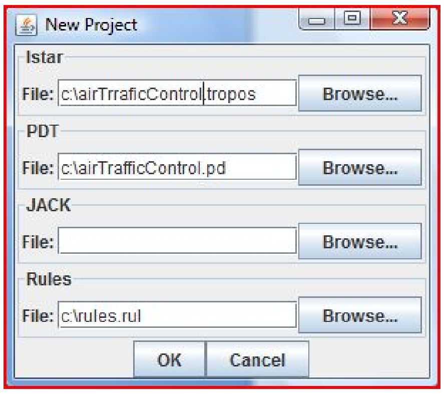


Figure 4.32 Creating a New Project

When the user selects the option ***New Project***, a new window opens where the user can enter the file name of the models and the rule used by the tool to generate traceability relations and to identify missing elements. For instance, Figure 4.33 shows a new project created where the user has entered ***airTrafficControl.tropos***, ***airTrafficControl.pd***, and ***rules.rul*** file names to generate traceability relations and identify missing elements between ***airTrafficControl.tropos*** (*i** model) and ***airTrafficControl.pd*** (Prometheus model) using the rules defined in the ***rules.rul*** file.


New Project

Istar
File: c:\airTrraficControl.tropos

PDT
File: c:\airTrafficControl.pd

JACK
File:

Rules
File: c:\rules.rul

Figure 4.33 New Project window

To generate traceability relations and to identify missing elements the user can select ***Run*** menu option followed by the menu item ***Run*** (see Figure 4.34)***.*** The prototype tool executes the project rules and it creates an ***output.xml*** file that contains traceability relations and the missing elements.

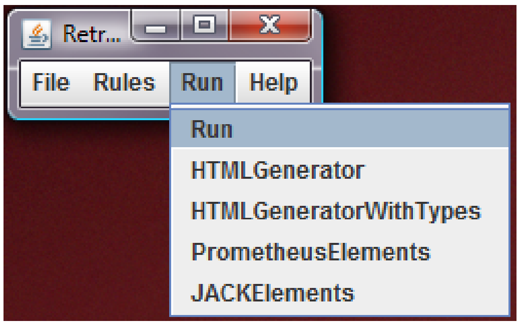


Figure 4.34 Creating traceability relations and identifying missing elements

For instance, Figure 4.35 shows part of an ***output.xml*** file that contains the traceability relation and missing information created by the prototype tool.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Traceability>
    <TraceabilityRelation
       ruleID="rule1"
       degreeOfCompleteness="100"
       type="overlaps">
      <Element
        id="_vdOtUCQwEd6fbcmFsKI3Cw"
        name="Find Best Landing Time for an Aircraft"
        doc="file:///C:/airTrafficControl.tropos"
        type="SD Goal"/>
      <Element
        id="49"
        name="Find Best Land Time for an Aircraft"
        doc="file:///C:/airTraffic.pd"
        type="Prometheus Goal"/>
    </TraceabilityRelation>
    <TraceabilityRelation
       ruleID="rule1"
       degreeOfCompleteness="100"
       type="overlaps">
     <Element
       id="_vdOtUCQwEd6fbcmFsKI3Cw"
       name="Find Best Landing Time for an Aircraft"
       doc="file:///C:/airTrafficControlEnvironment3.tropos"
       type="SD Goal"/>
     <Element
       id="113"
       name="Landing"
       doc="file:///C:/airTraffic.pd"
       type="Prometheus Goal"/>
</TraceabilityRelation>
....
</Traceability>
```

Figure 4.35 output.xml file

The tool allows the user to generate html reports from the *output.xml* file that contains the traceability relations. The user can select the menu item to generate a simple html report (see Figure 4.36) that contains the id of rule, the type of the element source and target, and the name of the source and target elements.

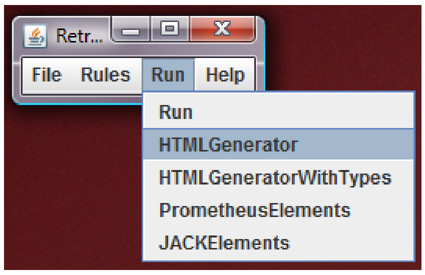


Figure 4.36 – HTML Generator sub-menu item

Figure 4.37 shows part of the output generated by the tool when the user has selected the *HTMLGenerator* menu item after the *output.xml* file has been created (see Figure 4.38).

```
Root element of the doc is Traceability
Total no of traceability relations : 85
<table class="sofT" cellspacing="0">
<td colspan="3" class="TopHed"> Traceability Relations Types between
Prometheus and JACK Artefacts</td>
<tr>
  <td class="helpHed">Rule ID</td>
  <td class="helpHed">SD Goal</td>
  <td class="helpHed">Goal</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">Allocate Runway Slot</td>
  <td class="helpBod">Allocate Runway Slot</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">Find Best Landing Time for an Aircraft</td>
  <td class="helpBod">Landing</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">Find Best Landing Time for an Aircraft</td>
  <td class="helpBod">Find Best Land Time for an Aircraft</td>
</tr>
<tr>
  <td class="helpHed">Rule ID</td>
  <td class="helpHed">SR Goal</td>
  <td class="helpHed">Goal</td>
</tr>
<tr>
  <td class="helpBod">rule3a</td>
  <td class="helpBod">Allocate Runway Slot</td>
  <td class="helpBod">Allocate Runway Slot</td>
</tr> ...
```

Figure 4.37 Simple HTML Report

The user can combine the template shown in Figure 4.38 together with the retratos.css file (see Figure 4.39) to present the result generated by the HTMLGenerator.

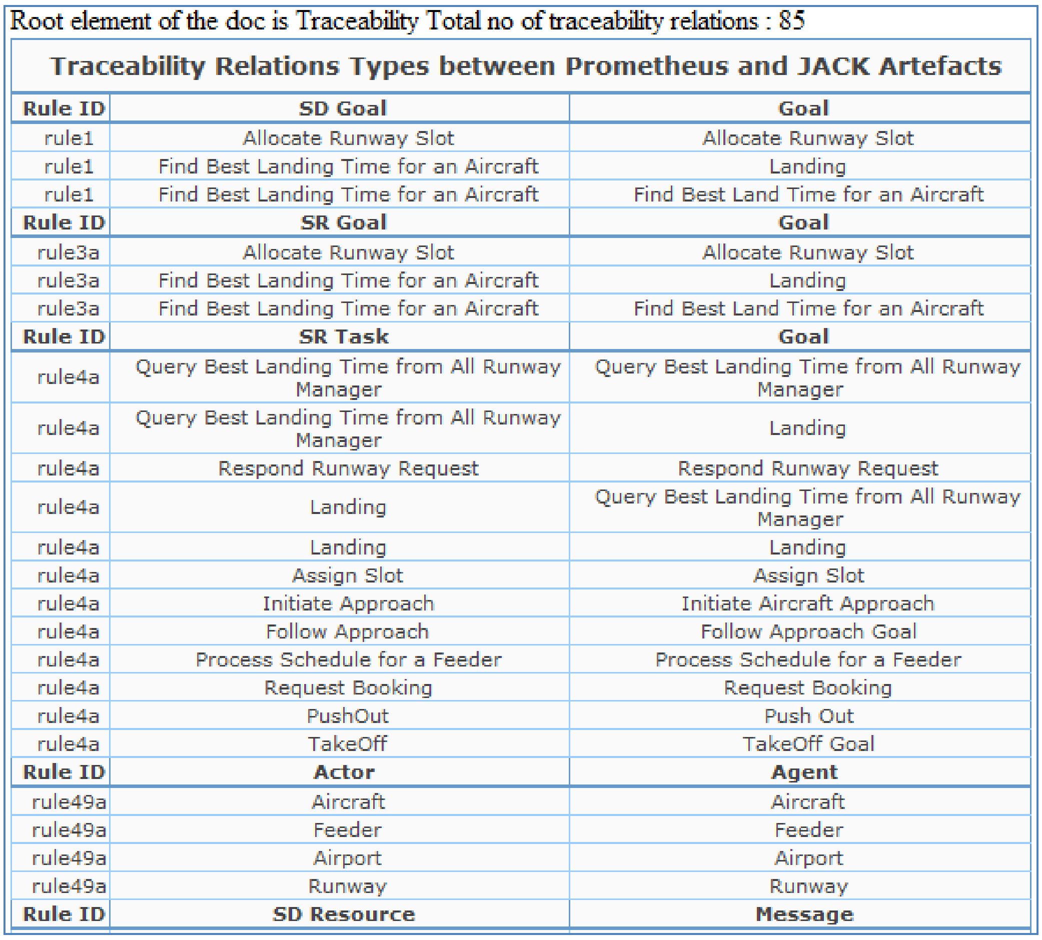
```
<html>
<head>
<link rel=stylesheet href="./retratos.css">
<title>Completeness Checking Rules between i* and
Prometheus Models</title>
</head>
<body>
                    // To add file generated by the HTMLGenerator tool
</body>
</html>
```

Figure 4.38 HTML Template

```
table.helpT {
text-align: center; font-family: Verdana; font-weight: normal;
font-size: 11px; color: #404040; width: 500px;
background-color: #fafafa; border: 1px #6699CC solid;
border-collapse: collapse; border-spacing: 0px;}

td.TopHed {
border-bottom: 2px solid #6699CC; border-left: 1px solid #6699CC;
text-align: center; text-indent: 5px; font-family: Verdana;
font-weight: bold; font-size: 14px; color: #404040;
padding-top: 6px; padding-bottom: 6px;}

td.helpHed {
border-bottom: 2px solid #6699CC; border-left: 1px solid #6699CC;
text-align: center; text-indent: 5px; font-family: Verdana;
font-weight: bold; font-size: 11px; color: #404040; }

td.helpHedLeft {
border-bottom: 1px solid #6699CC; border-left: 1px solid #6699CC;
text-align: center; text-indent: 5px; font-family: Verdana;
font-weight: bold; font-size: 11px; color: #404040; }
```

Figure 4.39 – retratos.css file

For instance, Figure 4.40 shows when the html report generated by the tool (see Figure 4.39) is included in the HTML template (see Figure 4.38).


Root element of the doc is Traceability Total no of traceability relations : 85

Traceability Relations Types between Prometheus and JACK Artefacts		
Rule ID	**SD Goal**	**Goal**
rule1	Allocate Runway Slot	Allocate Runway Slot
rule1	Find Best Landing Time for an Aircraft	Landing
rule1	Find Best Landing Time for an Aircraft	Find Best Land Time for an Aircraft
Rule ID	**SR Goal**	**Goal**
rule3a	Allocate Runway Slot	Allocate Runway Slot
rule3a	Find Best Landing Time for an Aircraft	Landing
rule3a	Find Best Landing Time for an Aircraft	Find Best Land Time for an Aircraft
Rule ID	**SR Task**	**Goal**
rule4a	Query Best Landing Time from All Runway Manager	Query Best Landing Time from All Runway Manager
rule4a	Query Best Landing Time from All Runway Manager	Landing
rule4a	Respond Runway Request	Respond Runway Request
rule4a	Landing	Query Best Landing Time from All Runway Manager
rule4a	Landing	Landing
rule4a	Assign Slot	Assign Slot
rule4a	Initiate Approach	Initiate Aircraft Approach
rule4a	Follow Approach	Follow Approach Goal
rule4a	Process Schedule for a Feeder	Process Schedule for a Feeder
rule4a	Request Booking	Request Booking
rule4a	PushOut	Push Out
rule4a	TakeOff	TakeOff Goal
Rule ID	**Actor**	**Agent**
rule49a	Aircraft	Aircraft
rule49a	Feeder	Feeder
rule49a	Airport	Airport
rule49a	Runway	Runway
Rule ID	**SD Resource**	**Message**

Figure 4.40 HTML Report using HTML template and retratos.css file

The user can also select the menu item to generate html report with the type of relations (see Figure 4.41) that contains the id of the rule, the type of the element source and target, and the name of the source and target elements.

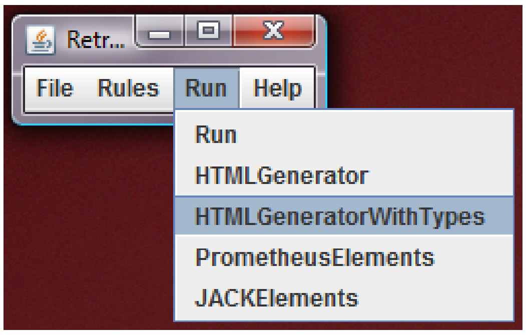


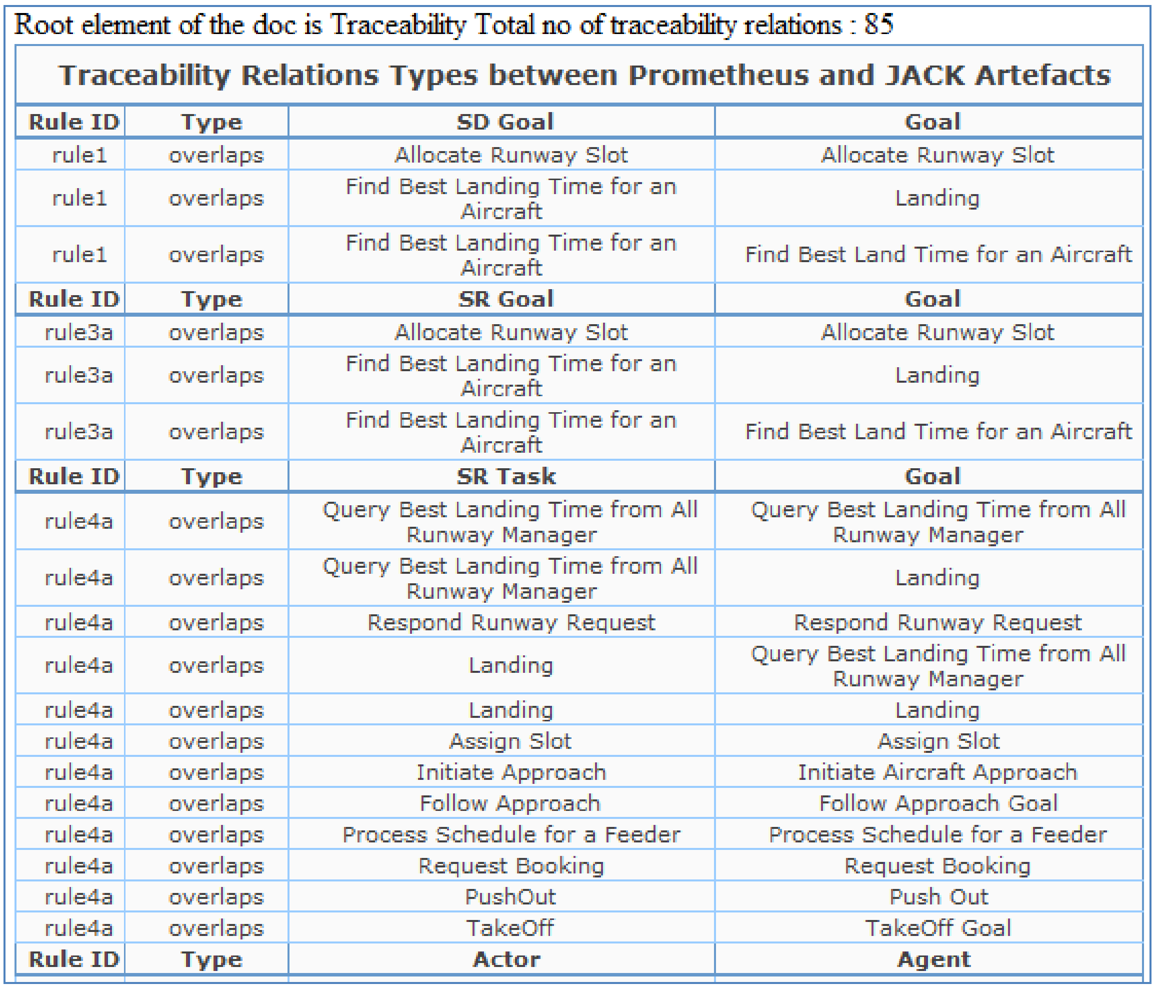
Figure 4.41 HTMLGeneratorWith Types menu item

Figure 4.42 shows part of the output generated by the tool when the user has selected the ***HTMLGenerator*** menu item after the ***output.xml*** file has been created (see Figure 4.35).

```
Root element of the doc is Traceability
Total no of traceability relations : 85
<table class="sofT" cellspacing="0">
<td colspan="4" class="TopHed">
Traceability Relations Types between Prometheus and JACK Artefacts
</td>
<tr>
  <td class="helpHed">Rule ID</td>
  <td class="helpHed">Type</td>
  <td class="helpHed">SD Goal</td>
  <td class="helpHed">Goal</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">overlaps</td>
  <td class="helpBod">Allocate Runway Slot</td>
  <td class="helpBod">Allocate Runway Slot</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">overlaps</td>
  <td class="helpBod">Find Best Landing Time for an Aircraft</td>
  <td class="helpBod">Landing</td>
</tr>
<tr>
  <td class="helpBod">rule1</td>
  <td class="helpBod">overlaps</td>
  <td class="helpBod">Find Best Landing Time for an Aircraft</td>
  <td class="helpBod">Find Best Land Time for an Aircraft</td>
</tr>
<tr>
  <td class="helpHed">Rule ID</td>
  <td class="helpHed">Type</td>
  <td class="helpHed">SR Goal</td>
  <td class="helpHed">Goal</td>
</tr><tr>
  <td class="helpBod">rule3a</td>
  <td class="helpBod">overlaps</td>
  <td class="helpBod">Allocate Runway Slot</td>
  <td class="helpBod">Allocate Runway Slot</td>
</tr>
```

Figure 4.42 HTML Report with Types

The user can also combine the template shown in Figure 4.50 together with the ***retratos.css*** file (see Figure 4.39) to present the result generated by the ***HTMLGeneratorWithTypes***. Figure 4.43 shows when the html report generated by the tool (see Figure 4.42) is included in the HTML template (see Figure 4.38).



Root element of the doc is Traceability Total no of traceability relations : 85

Traceability Relations Types between Prometheus and JACK Artefacts

Rule ID	Type	SD Goal	Goal
rule1	overlaps	Allocate Runway Slot	Allocate Runway Slot
rule1	overlaps	Find Best Landing Time for an Aircraft	Landing
rule1	overlaps	Find Best Landing Time for an Aircraft	Find Best Land Time for an Aircraft
Rule ID	**Type**	**SR Goal**	**Goal**
rule3a	overlaps	Allocate Runway Slot	Allocate Runway Slot
rule3a	overlaps	Find Best Landing Time for an Aircraft	Landing
rule3a	overlaps	Find Best Landing Time for an Aircraft	Find Best Land Time for an Aircraft
Rule ID	**Type**	**SR Task**	**Goal**
rule4a	overlaps	Query Best Landing Time from All Runway Manager	Query Best Landing Time from All Runway Manager
rule4a	overlaps	Query Best Landing Time from All Runway Manager	Landing
rule4a	overlaps	Respond Runway Request	Respond Runway Request
rule4a	overlaps	Landing	Query Best Landing Time from All Runway Manager
rule4a	overlaps	Landing	Landing
rule4a	overlaps	Assign Slot	Assign Slot
rule4a	overlaps	Initiate Approach	Initiate Aircraft Approach
rule4a	overlaps	Follow Approach	Follow Approach Goal
rule4a	overlaps	Process Schedule for a Feeder	Process Schedule for a Feeder
rule4a	overlaps	Request Booking	Request Booking
rule4a	overlaps	PushOut	Push Out
rule4a	overlaps	TakeOff	TakeOff Goal
Rule ID	**Type**	**Actor**	**Agent**

Figure 4.43 HTML Report with types using HTML template and retratos.css file

The tool provides functionalities to visualize rules defined in the project and to create new rules. The user can select the menu item ***Rules->Create New Rules->IstarPrometheusRule*** to create a new rule to generate traceability relations between *i** and Prometheus elements and to identify missing elements (see Figure 4.44).

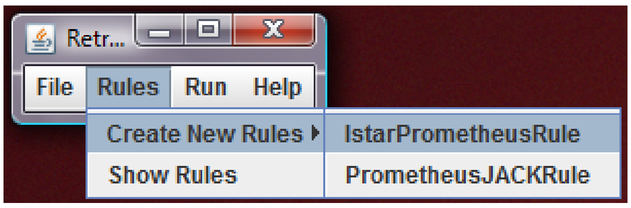
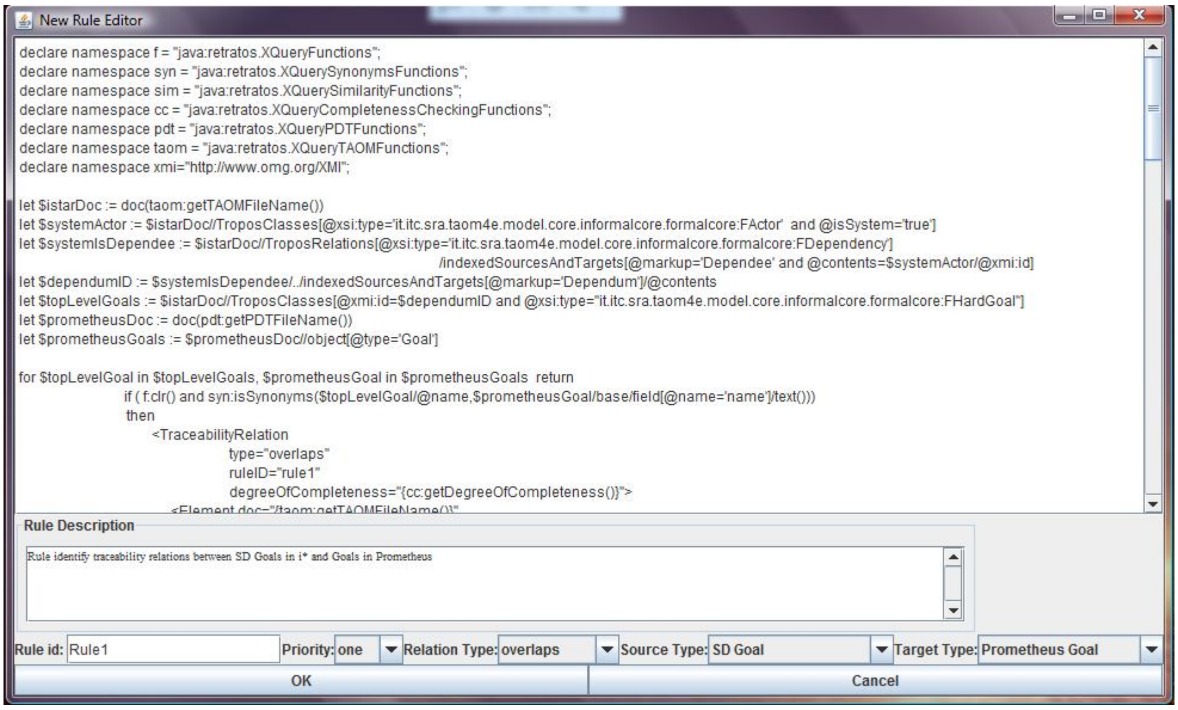


Figure 4.44 IstarPrometheusRule menu item

Figure 4.45 shows the rule editor used to create Rule1 to identify traceability relations between SD goals in *i** and goals in Prometheus with priority equal to one.



```
declare namespace f = "java:retratos.XQueryFunctions";
declare namespace syn = "java:retratos.XQuerySynonymsFunctions";
declare namespace sim = "java:retratos.XQuerySimilarityFunctions";
declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
declare namespace pdt = "java:retratos.XQueryPDTFunctions";
declare namespace taom = "java:retratos.XQueryTAOMFunctions";
declare namespace xmi="http://www.omg.org/XMI";

let $istarDoc := doc(taom:getTAOMFileName())
let $systemActor := $istarDoc//TroposClasses[@xsi:type='it.itc.sra.taom4e.model.core.informalcore.formalcore:FActor'  and @isSystem='true']
let $systemIsDependee := $istarDoc//TroposRelations[@xsi:type='it.itc.sra.taom4e.model.core.informalcore.formalcore:FDependency']
                                        /indexedSourcesAndTargets[@markup='Dependee' and @contents=$systemActor/@xmi:id]
let $dependumID := $systemIsDependee/../indexedSourcesAndTargets[@markup='Dependum']/@contents
let $topLevelGoals := $istarDoc//TroposClasses[@xmi:id=$dependumID and @xsi:type="it.itc.sra.taom4e.model.core.informalcore.formalcore:FHardGoal"]
let $prometheusDoc := doc(pdt:getPDTFileName())
let $prometheusGoals := $prometheusDoc//object[@type='Goal']

for $topLevelGoal in $topLevelGoals, $prometheusGoal in $prometheusGoals  return
          if ( f:clr() and syn:isSynonyms($topLevelGoal/@name,$prometheusGoal/base/field[@name='name']/text()))
          then
             <TraceabilityRelation
                    type="overlaps"
                    ruleID="rule1"
                    degreeOfCompleteness="{cc:getDegreeOfCompleteness()}">
               <Element doc="{taom:getTAOMFileName()}"
```

Rule Description

Rule identify traceability relations between SD Goals in i* and Goals in Prometheus

Rule id:	Rule1	Priority:	one	Relation Type:	overlaps	Source Type:	SD Goal	Target Type:	Prometheus Goal

Figure 4.45 IstarPrometheus rule editor

The user can select the menu item ***Rules->Create New Rules->PrometheusJACKRule*** to create a new rule to generate traceability relations between *i** and Prometheus elements and to identify missing elements (see Figure 4.46).

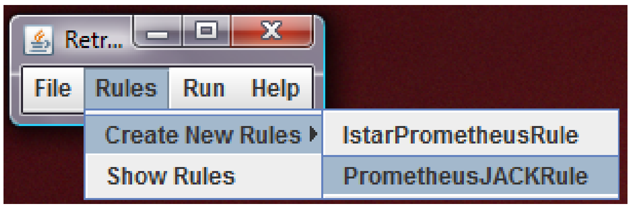
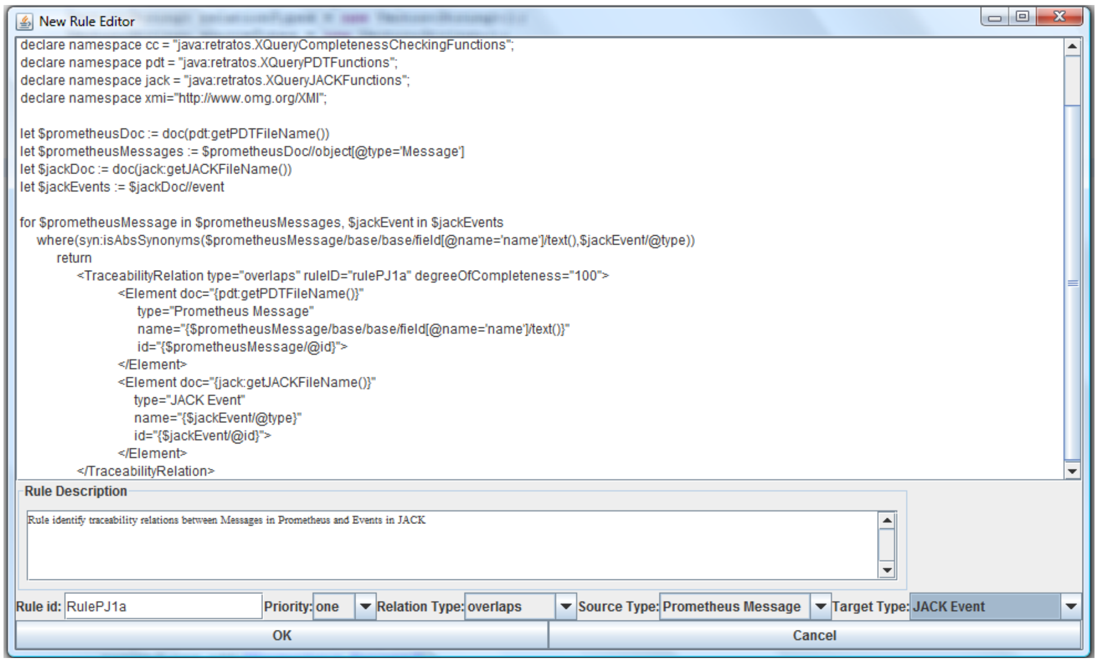


Figure 4.46 PrometheusJACKRule menu item

Figure 4.47 shows the rule editor used to create the ***Rule1*** to identify traceability relations between SD goals in *i** and goals in Prometheus with priority equal to one.



```
declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
declare namespace pdt = "java:retratos.XQueryPDTFunctions";
declare namespace jack = "java:retratos.XQueryJACKFunctions";
declare namespace xmi="http://www.omg.org/XMI";

let $prometheusDoc := doc(pdt:getPDTFileName())
let $prometheusMessages := $prometheusDoc//object[@type='Message']
let $jackDoc := doc(jack:getJACKFileName())
let $jackEvents := $jackDoc//event

for $prometheusMessage in $prometheusMessages, $jackEvent in $jackEvents
   where(syn:isAbsSynonyms($prometheusMessage/base/base/field[@name='name']/text(),$jackEvent/@type))
      return
         <TraceabilityRelation type="overlaps" ruleID="rulePJ1a" degreeOfCompleteness="100">
                <Element doc="{pdt:getPDTFileName()}"
                    type="Prometheus Message"
                    name="{$prometheusMessage/base/base/field[@name='name']/text()}"
                    id="{$prometheusMessage/@id}">
                </Element>
                <Element doc="{jack:getJACKFileName()}"
                   type="JACK Event"
                   name="{$jackEvent/@type}"
                   id="{$jackEvent/@id}">
                </Element>
         </TraceabilityRelation>
```

Rule Description

Rule identify traceability relations between Messages in Prometheus and Events in JACK

Rule id: RulePJ1a Priority: one Relation Type: overlaps Source Type: Prometheus Message Target Type: JACK Event

Figure 4.47 PrometheusJACK rule editor

The user can select the menu item ***Rules->Show Rules*** (see Figugre 4.48) to visualize the rules used in the project.

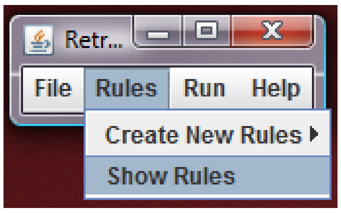
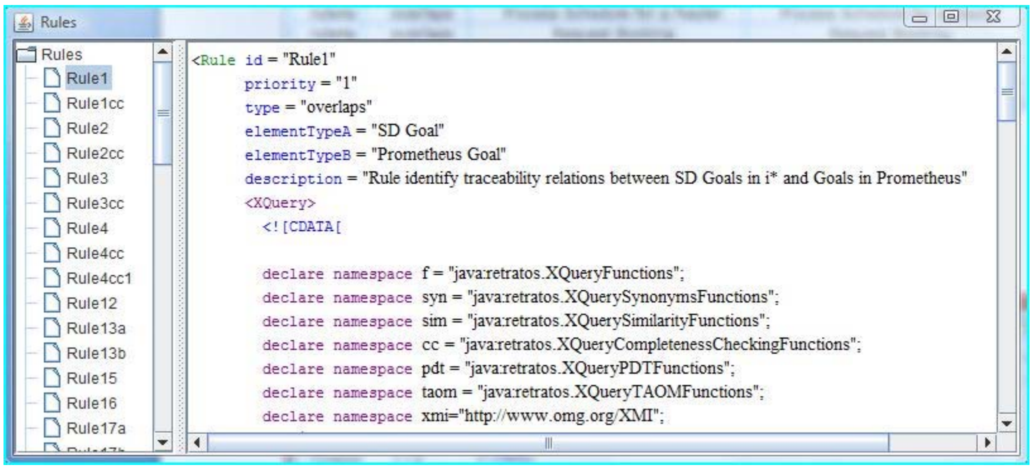


Figure 4.48 Show Rules menu item

Figure 4.49 shows the rule viewer that shows the ***Rule1*** to identify traceability relations between SD goals in *i** and goals in Prometheus with priority equal to one.



```
<Rule id = "Rule1"
      priority = "1"
      type = "overlaps"
      elementTypeA = "SD Goal"
      elementTypeB = "Prometheus Goal"
      description = "Rule identify traceability relations between SD Goals in i* and Goals in Prometheus"
      <XQuery>
        <![CDATA[

        declare namespace f = "java:retratos.XQueryFunctions";
        declare namespace syn = "java:retratos.XQuerySynonymsFunctions";
        declare namespace sim = "java:retratos.XQuerySimilarityFunctions";
        declare namespace cc = "java:retratos.XQueryCompletenessCheckingFunctions";
        declare namespace pdt = "java:retratos.XQueryPDTFunctions";
        declare namespace taom = "java:retratos.XQueryTAOMFunctions";
        declare namespace xmi="http://www.omg.org/XMI";
```

Figure 4.49 Rule Viewer

4.5 Discussion

Our work is similar to the work in (Spanoudakis, et al., 2004), (Jirapanthong, et al., 2005), (Jirapanthong, et al., 2009) given that it is based on the use of rules to generate traceability relations between software artefacts. However, our work differs from these approaches with respect to (i) the domain to which the work is applied (i.e., multi-agent systems), (ii) the way that rules are specified and created, and (iii) the fact that it covers several phases of the software development life-cycle (i.e., early requirements, design, and implementation phases). The types of traceability relations used in our work are different from the ones suggested in (Spanoudakis, et al., 2004) and (Jirapanthong, et al., 2009) given that they are concerned with software models generated when using *i** framework, Prometheus methodology, and JACK language. Moreover, our work does not make use of grammatical roles of the terms that exist in the software models since the documents of our concern do not have textual descriptions in the form of sentences and paragraphs, as found in requirements specifications. Instead, our work relies on rules that are based on the semantics of the artefacts and their relations. We use WordNet dictionary to implement the synonym function as in Jirapanthong's (Jirapanthong, et al., 2009) work. However, in Jirapanthong's work a term is passed to a function that retrieves a list of synonyms for that term, while in our work we use a function to compare if all the terms used in an artefact, or a property, are synonyms to all the terms used to define another artefact or property. Our work also uses rules expressed in XML with embedded code in XQuery as in the case of (Jirapanthong, et al., 2009). However, in our framework, the rules also support completeness checking that can indentify missing elements in the models and can assist with the completion of the models and fix discrepancies of names initially given to the elements. Other differences are concerned with the use of a special attribute called degreeOfCompleteness in traceability relations to measure the level of confidence on the generated relations; and the use of XQuery to define the action to be executed if the condition part of a rule is satisfied. Furthermore, we believe that the use of XQuery to specify rules is simpler than the use of the XML-based mark-up language adopted in the work in (Spanoudakis, et al., 2003), (Spanoudakis, et al., 2004).

Our work differs from the work in (Reiss, 2006) in various ways: (a) our rules identify specific types of traceability relations and missing elements, instead of general consistency rules; (b) our rules are described in an XML-based language in order to assist with the manipulation of XML elements; and (c) our artefacts are concerned with agent oriented software models, instead of object-oriented models.

The work presented in (Alves-Foss, et al., 2002) is similar to our approach given that our approach also uses XML to represent elements and traceability relations. Our approach uses XQuery to define traceability rules while in Alves-Foss' approach, rules are represented using XSL.

4.6 Summary

In this chapter we have presented our rule-based traceability framework to support automatic generation of different types of traceability relations and identification of missing elements between different types of software models generated during the development of Agent Oriented Systems. We have presented an overview of our framework, the different types of rules used to support the generation of traceability relations and identification of missing elements, extra functions that we have created to support the rules, and a description of a tool that we have developed.

Chapter 5 - Evaluation and Results

We evaluate our approach through three case studies namely (a) Automatic Teller Machine (ATM), (b) Air Traffic Control Environment (ATCE), and (c) Electronic Bookstore (EB) systems.

The selection of the case studies was based on the need to have a large size case study to demonstrate scalability and the effectiveness of the tool in terms of recall and precision, to have a medium size case study to demonstrate completeness checking and to have a small size case to validate the rules. Initially, we identified the EB case study presented in (Padgham, et al., 2004) that is a typical example of commerce electronic application. The ATCE case study is a typical application of multi-agent systems. The ATM is small case study in a well known application area.

The ATM system is composed of design models in Prometheus and code implemented in JACK. The code used came from examples provided by AOS (http://www.agent-software.com.au/products/jack/documentation_and_instructi/index.html). The Prometheus model was build based on this code. The ATM is small case study that helps to demonstrate the approach and show the approach is working.

The ATCE is composed of *i**, Prometheus models and JACK code. The code used came from examples provided by AOS (http://www.agent-software.com.au/products/jack/documentation_and_instructi/index.html). The Prometheus model was build based on these code and the *i** model was build based on the Prometheus model and documentation of Air Traffic Control Systems (Ljungberg, et al., 1992). The ATCE was selected to demonstrate how the approach can use the information about the missing elements to fix discrepancies between names given to elements in the different documents and to improve completeness between software artefacts created during the development of multi-agent systems.

The EB systems is composed of *i** models, Prometheus design models, and JACK code. The Prometheus model was based on the case study presented in (Padgham, et al., 2004). The *i** model was produced based on the general knowledge how a bookstore works and the JACK code was created based on the Prometheus model. The EB system is a medium-size scale case study that was used to demonstrate that the approach can automatically generate traceability relations.

We describe below the criteria for evaluating the work, an overview of the case studies, and the results of the evaluation for each case study.

5.1 Criteria for Evaluation

Our work has been evaluated in order to demonstrate the hypotheses presented in Chapter 1. More specifically, the work has been evaluated to demonstrate that the approach

(a) can automatically generate traceability relations for documents created during the development of multi-agent systems

(b) can automatically identify missing elements in the documents created during the development of multi-agent systems

(c) can use the information about missing elements to fix discrepancies between names given to elements in the different documents and to improve completeness between software artefacts created during the development of multi-agent systems

(d) can use the information about missing elements to improve the number of traceability relations identified by the tool in other iterations

In order to evaluate hypotheses (a), we measured the recall and precision of the traceability relations generated by the tool. The use of recall and precision measures to evaluate tools to support automatic generation of traceability relations have been advocated in the literature (Marcus, et al., 2003), (Marcus, et al., 2005), (Spanoudakis, et al., 2004), (De Lucia, et al., 2004). We used the following standard defection of recall and precision given in (Marcus, et al., 2003).

$$\text{Precision} = \frac{|T \cap M|}{M}$$

$$\text{Recall} = \frac{|T \cap M|}{T}$$

where,

T is the number of traceability relations identified by the tool;

M is the number of traceability relations identified manually by the user (we assume that the user find all correct relations);

$|T \cap M|$ is the number of common traceability relations identified by the tool and by the user.

In order to evaluate hypotheses (b), we measure the recall and precision of the models used in our case study, identify the missing elements, amend the models used in our case studies based on the identified missing elements, and measure the recall and precision of the amended models in order to verify if there have been changes in the recall and precision measures.

In order to evaluate (c), we present some examples where the information about missing elements was used to fix discrepancies between names given to elements in the different abstract levels of documentation and to improve completeness between software artefacts created during the development of multi-agent systems.

In order to evaluate (d), we complete the models using the information about missing elements and create a new set of the models. We use the tool to identify traceability relations in the new created models, and measure recall and precision. We compare the results of recall and precision with the previous results obtained for recall and precision.

In the next sections we present the Automatic Teller Machine, Air Traffic Control Environment, and Electronic Bookstore case studies, with the results of their evaluation.

5.2 Automatic Teller Machine

5.2.1 Overview of the Case Study

This subsection describes the development of a multi-agent system to implement the Automated Teller Machine (ATM) used as one of the case study in this thesis report. The Automated Teller Machine (ATM) allows customers to carry out bank transactions without the assistance of a teller. Examples of these transactions are: withdraw of cash, change of PIN, execution of a payment, checking of account's balance, printing statement, and transfer of money. The customer needs to insert a card in the ATM machine and enter a PIN code to use one of the services provided by the ATM machine. When the customer inserts the card the system reads the card details and shows a screen asking for a PIN number. The customer enters the PIN number which is validated by the system. If the PIN number is correct the system shows a screen with the services available for the customer. If the PIN number is not correct the system requests the number again to the customer up to three times, when the card is retained by the system.

When the customer selects withdraw cash option, the system shows withdraw cash screen. The customer enters the amount of money that he/she would like to withdraw and then the system processes the cash withdraw. The system requests the Bank authorization to execute the cash withdraw. If the Bank approves the cash withdraw, the ATM machine dispenses the amount of cash requested by the customer and prints a receipt. If the Bank does not approve the cash withdraw, the ATM machine shows a message given details why the cash withdraw was not authorized.

If the customer selects to change the PIN number, the ATM machine shows a screen where the customer can enters a new PIN number. After the customer enters the new PIN number, the ATM sends the new PIN number to the Bank.

If the customer selects to make a payment, the ATM shows a screen where the customer can enter details about the payment. The ATM sends details about the payment to the Bank to execute the payment.

If the customer selects the balance account, the ATM requests the balance to the Bank and then shows the balance on the screen.

If the customer selects to print a statement, the ATM requests the transactions done by the customer to the Bank and then prints the details of the transactions.

If the customer selects to transfer money, the ATM shows a screen where the customer can enter details about the account to where transfer the money and the amount to be transferred. The ATM sends the details about the money transfer to the Bank to execute the transfer.

Support Staff of the Bank periodically performs maintenance on the ATM machine. The Support Staff are responsible to replace tonner when necessary, deposit cash into the ATM machine when a low quantity of cash is received, add paper for the printer when alerted, and perform other types of printer maintenance when necessary.

The ATM case study used in our work was developed using (a) Prometheus methodology to create the system specification, analysis, and design models, and (b) JACK language to implement the system.

The ATM case study was used in two different experiments. Firstly, we used the tool to measure recall and precision and to execute completeness checking between Prometheus models and JACK code. Secondly, we rectified the models based on the results of the completeness checking execution and used the tool to measure new recall and precision of the rectified models.

The next sections describe in detail the development of the ATM case study and its evaluation. The ATM case study used in our work was developed using Prometheus methodology to create the system specification, analysis and design models; and JACK language to implement the system.

5.2.2 Artefacts

The Automatic Teller Machine (ATM) application case study consists of a multi-agent system composed of two agents, namely ATM and Bank agents. It is composed of the following Prometheus diagrams: (i) one Goal Overview diagram, (ii) one System Overview diagram, and (iii) two Agent Overview Diagram.

Table 5.1 shows the number of elements vs. types of elements in Prometheus used in the ATM case study.

Type of the element	Number of Elements
Agent	2
Action	5
Data	2
Goal	4
Message	3
Percept	3
Plan	4

Table 5.1 ATM elements in Prometheus

Table 5.2 shows the number of elements vs. types of elements in JACK used in the ATM case study.

Type of the element	Number of Elements
Agent	2
BeliefSet	5
Event	2
Plan	4

Table 5.2 ATM elements in JACK

The full representations of Prometheus diagrams and JACK code in XML for this case study are shown in Appendix B.

5.2.3 Evaluation

The ATM case study was used to evaluate our work in order to:

- measure recall and precision of our tool;
- identify missing elements;
- show how the identified missing elements can support the development of the system by increasing its recall.

It is important highlight that this case study was used to demonstrate the performance of the tool in discovering and correcting defects in the models.

In order to demonstrate the above, we have (i) identified traceability relations between the models manually, (ii) identified traceability relations between the models using our tool, (iii) measured recall and precision by comparing the results in (i) and (ii), (iv) identified missing elements using our tool, (v) used the results in (iv) to manually complete the models creating a new set of the models, (vi) used the tool to identify traceability relations and missing elements in the new models created in (iv), and (vii) measured the recall and precision of the results in (vi).

Our experiment has been conducted using 47 traceability rules to identify traceability relations between Prometheus elements and JACK code and 16 completeness checking rules to identify missing elements between Prometheus elements and JACK code, for the ATM case study.

Table 5.3 shows the results of our experiments for (i), (ii) and (iii) activities.

Number of relations identified by the engine |T|	31
Number of relations identified by the user |M|	64
Number of missing relations	33
Number of incorrect relations	0
Number of relations that intercept T and M	31
Recall	50%
Precision	100%

Table 5.3 Results of experiments for the ATM case study

As shown in Table 5.3, the experiment achieved 50% of recall and 100% of precision. The table also shows that there were 33 missing traceability relations and 0 incorrect traceability relations. The tool also returns information about missing traceability relations due to the absence of the elements related to these relations in the models. The missing relations identified by the tool are shown in Table 5.4. The number of missing relations identified by the tool can be lower (i.e. 15) than the number of missing relations identified manually (i.e. 33).The data of the traceability relations identified manually and by the tool is shown in the Appendix C.

We used the information of the missing elements identified by the tool (see Table 5.4) to fix inconsistencies and complete the models. As shown in Table 5.4, there were missing relations between (a) JACK BeliefSet and Prometheus Data, (b) JACK Plan and Prometheus Plan, (c) Prometheus Plan and JACK Plan, (d) Prometheus Goal and JACK Agent, and (e) Prometheus Message and JACK Agent.

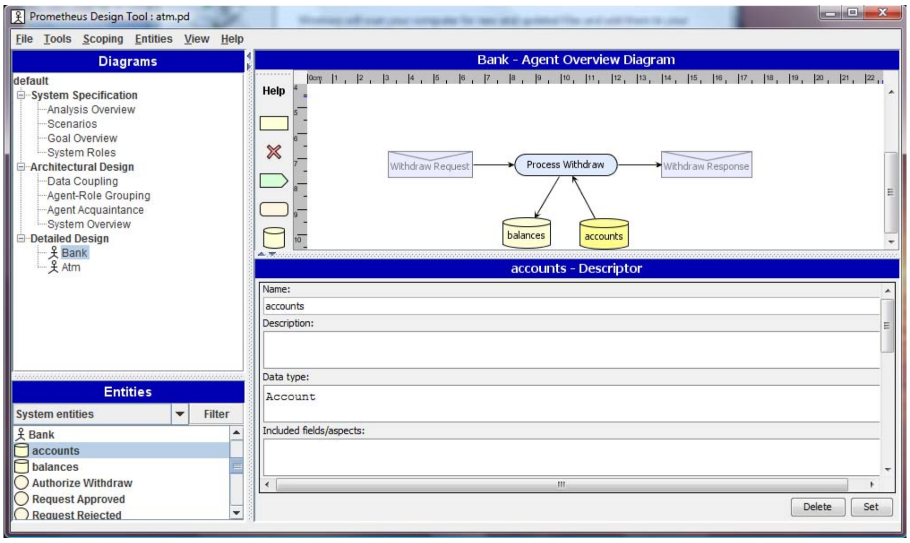
Rule ID	JACK BeliefSet	Prometheus Data
RulePJ2cc1	Accounts	
RulePJ2cc1	Balances	
Rule ID	**JACK Plan**	**Prometheus Plan**
RulePJ3cc1	ProcessWithdraw	
RulePJ3cc1	WithdrawApproved	
RulePJ3cc1	WithdrawCash	
RulePJ3cc1	WithdrawRejected	
Rule ID	**Prometheus Plan**	**JACK Plan**
RulePJ3cc2	Withdraw Approved	
RulePJ3cc2	Process Withdraw	
RulePJ3cc2	Withdraw Cash	
RulePJ3cc2	Withdraw Rejected	
Rule ID	**Prometheus Goal**	**JACK Agent**
RulePJ5cc1	Request Approved	
RulePJ5cc1	Request Rejected	
RulePJ5cc1	Withdraw Money	
RulePJ5cc1	Authorize Withdraw	
Rule ID	**Prometheus Message**	**JACK Agent**
RulePJ12cc1	Withdraw	

Table 5.4 Missing Information

More specifically, Table 5.4 shows that ***RulePJ2cc1*** rule identifies missing traceability relations between ***Accounts*** and ***Balances*** beliefSets in JACK and some data in Prometheus. We look what data in Prometheus could be related to ***Accounts*** beliefSet and conclude that ***Accounts*** beliefSet in JACK should be related to ***Accounts*** data in Prometheus. BeliefSets in JACK and data in Prometheus are related when the name of the beliefSet and the name of data are synonyms and ***included fields/aspects*** properties of the data is similar to the ***fields*** in the beliefSet. Figure 5.1 and 5.2 shows that ***Accounts*** beliefSet has ***account*** and ***pin*** fields and no ***included fields/aspects*** properties has been defined to the ***Accounts*** data .

```
<beliefSet id="b1" type="Accounts" extends="OpenWorld">
            <field declarationType="key" type="int" name="account"/>
            <field declarationType="value" type="int" name="pin"/>
...
</beliefSet>
```

Figure 5.1 Fields of the Accounts beliefSetAccounts Descriptor



Name:
accounts
Description:

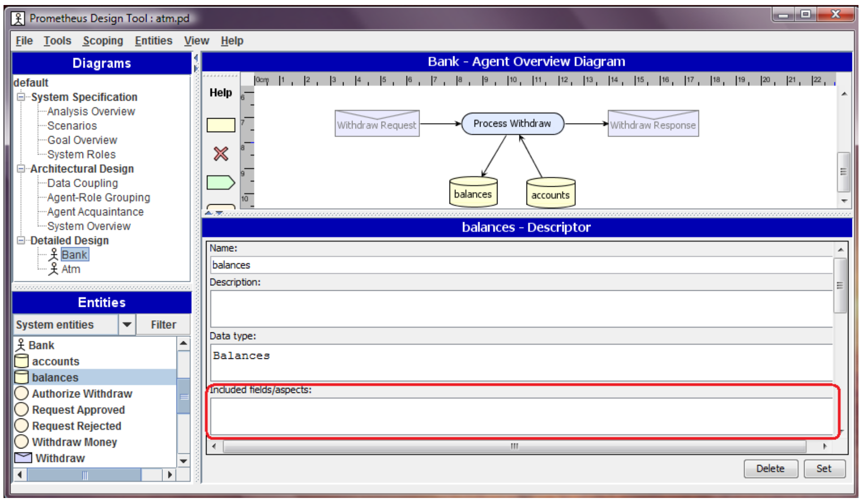
Data type:
Account
Included fields/aspects:

Figure 5.2 Accounts Descriptor

Similarly, we look what data in Prometheus could be related to ***Balances*** beliefSet. We concluded that ***Balances*** beliefSet in JACK should be related to ***Balances*** data in Prometheus. Figure 5.3 and 5.4 shows that ***Balances*** beliefSet has ***account*** and ***balance*** fields and no ***included fields/aspects*** properties has been defined to the ***Balances*** data.

```
<beliefSet id="b1" type="Balances" extends="OpenWorld">
  <field declarationType="key" type="int" name="account"/>
  <field declarationType="value" type="int" name="balance"/>
...
</beliefSet>
```

Figure 5.3 Balances beliefSet



Name: balances

Description:

Data type: Balances

Figure 5.4 Balances descriptor

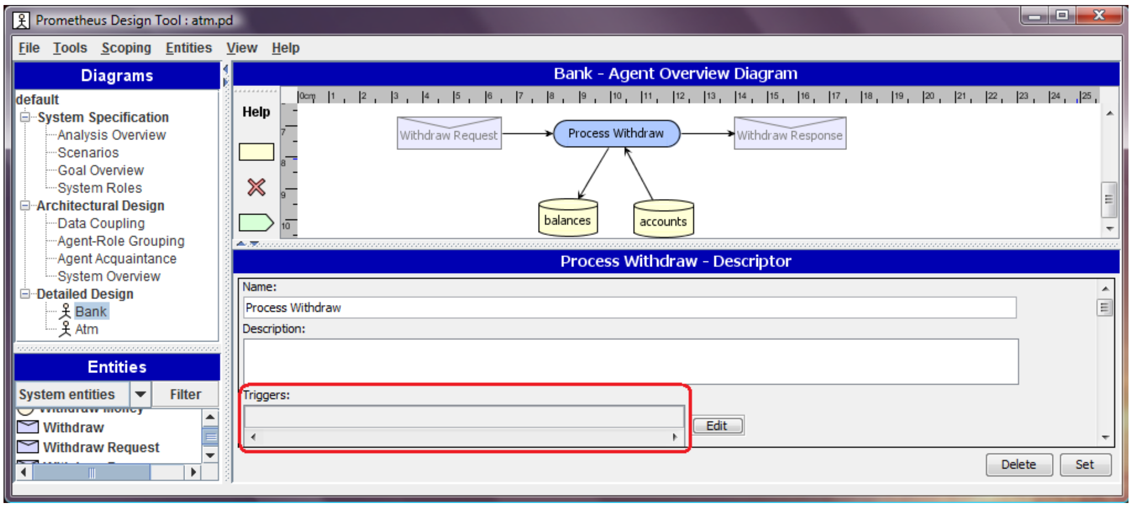
We added *accounts* and *balances* to the *included fields/aspects* to the *Balances* data in Prometheus.

Table 5.4 also shows (i) that *RulePJ3cc1* rule identifies that there are missing traceability relations between *ProcessWithdraw*, *WithdrawApproved*, *WithdrawCash* and *WithdrawRejected* plans in JACK and plans in Prometheus; and (ii) that *RulePJ3cc2* rule identifies that there are missing traceability relations between *Process Withdraw*, *Withdraw Approved*, *Withdraw Cash* and *Withdraw Rejected* plans in Prometheus and plans in JACK. Based on this information we can determine that there are missing traceability relations between *ProcessWithdraw* in JACK and *ProcessWithdraw* in Prometheus. Plans in JACK and plans in Prometheus are related when the name of the plan in JACK and the name of plan in Prometheus are synonyms, and the name of the element that triggers the plan in Prometheus and the name of the event that the plan in JACK handles are synonyms.

```
<plan id="p1" name="ProcessWithdraw" extends="Plan">
<handlesEvent type="WithdrawRequest" ref="event"/>
...
</plan>
```

Figure 5.5 ProcessWithdraw plan

We observed that ***ProcessWithdraw*** plan in JACK handles ***WithdrawRequest*** event (see Figure 5.5) while no trigger properties has been defined to the ***Process Withdraw*** plan in Prometheus (see Figure 5.6).



Name:
Process Withdraw
Description:
Triggers:

Figure 5.6 Process Withdraw descriptor

We added ***Withdraw Request*** message to the triggers properties of the ***Process Withdraw*** plan. Similarly, we identified that a traceability relations between ***ProcessWithdraw*** plan in JACK and ***ProcessWithdraw*** plan in Prometheus was missing. We identified that the ***WithdrawRequest*** plan in JACK handles ***WithdrawResponse*** event (see Figure 5.7) while no trigger properties has been defined to the ***Withdraw Approved*** plan in Prometheus (see Figure 5.12).

```
<plan id="p2" name="WithdrawApproved" extends="Plan">
   <handlesEvent type="WithdrawResponse" ref="event"/>
…
</plan>
```

Figure 5.7 WithdrawApproved plan

We added ***Withdraw Response*** message to the triggers properties of the ***Withdraw Approved*** plan.

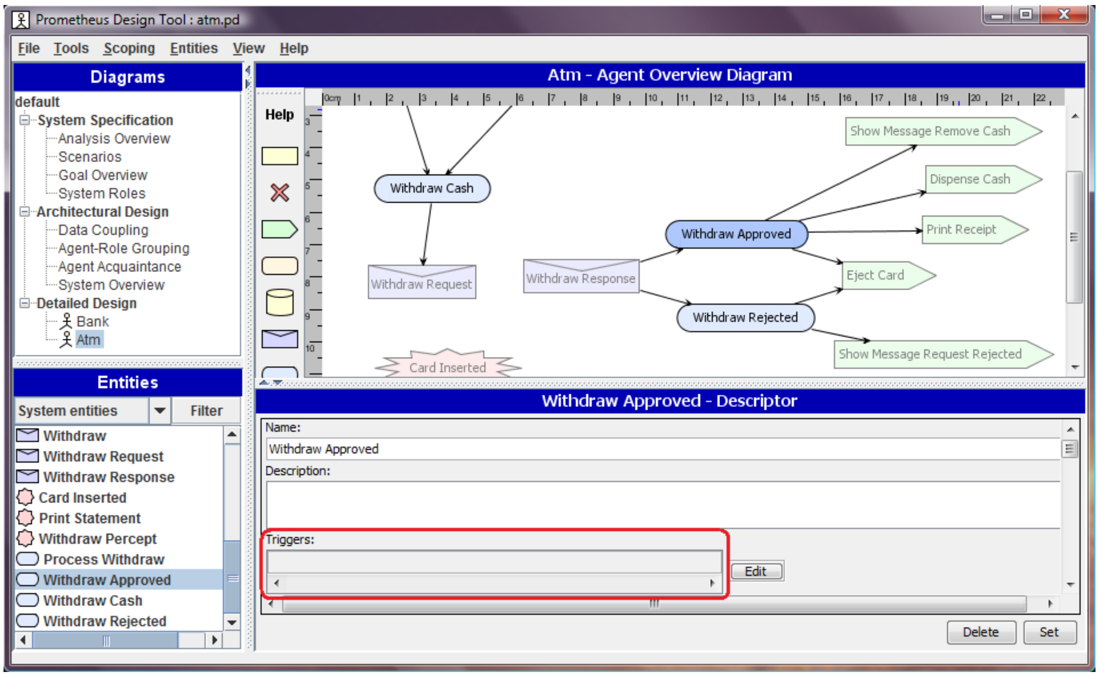
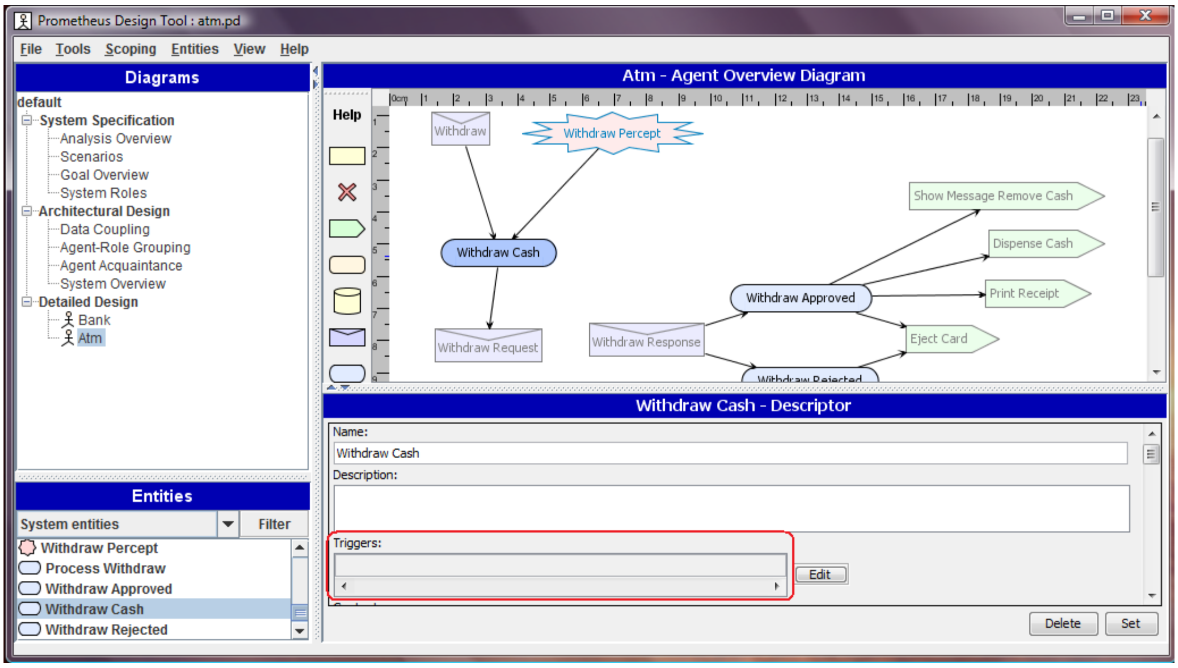


Figure 5.8 Withdraw Approved descriptor

The experiment also shows that a traceability relation was also missing between ***Withdraw Cash*** plan in JACK and ***Withdraw Cash*** in Prometheus. We found that ***WithdrawCash*** plan in JACK handles ***Withdraw*** event (see Figure 5.9) while no trigger properties has been defined to the ***Withdraw Cash*** plan in Prometheus (see Figure 5.10).

```
<plan id="p3" name="WithdrawCash" extends="Plan">
  <import>gui.AtmClient</import>
  <import>gui.AtmInterface</import>
  <handlesEvent type="Withdraw" ref="event"/>
...
</plan>
```

Figure 5.9 WithdrawCash plan



Name: Withdraw Cash

Description:

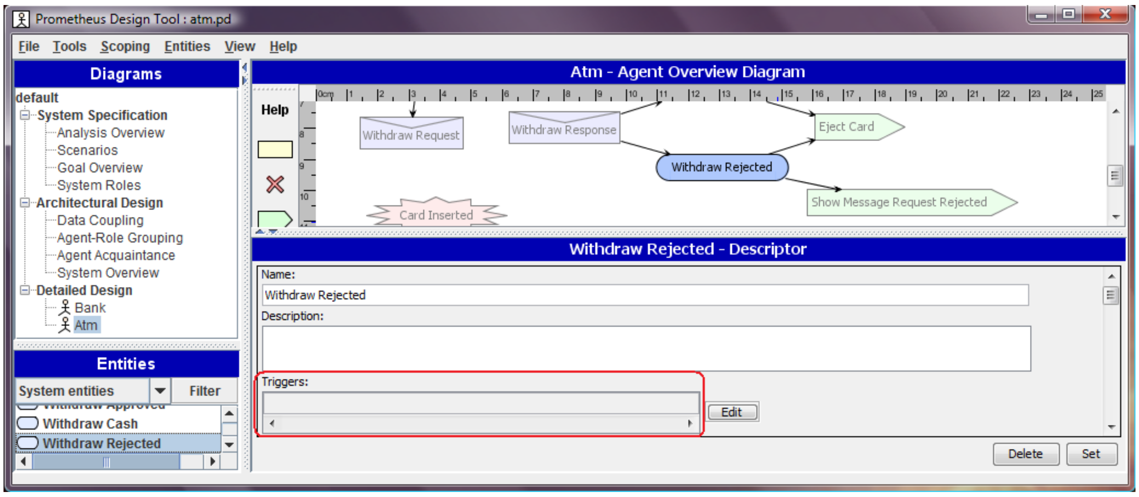
Triggers:

Figure 5.10 Withdraw Cash descriptor

We added Withdraw message to the triggers properties of the ***Withdraw Cash*** plan. A traceability relation was also missing between ***Withdraw Rejected*** plan in JACK and ***Withdraw Rejected*** plan in Prometheus. We found that ***WithdrawRejected*** plan in JACK handles ***WithdrawResponse*** event (see Figure 5.11) while no trigger properties has been defined to the ***Withdraw Rejected*** plan in Prometheus (see Figure 5.12).

```
<plan id="p4" name="WithdrawRejected" extends="Plan">
  <import>gui.AtmClient</import>
  <import>gui.AtmInterface</import>
  <handlesEvent type="WithdrawResponse" ref="event"/>
...
</plan>
```

Figure 5.11 WithdrawRejected plan



Name:
Withdraw Rejected
Description:

Figure 5.12 Withdraw Rejected descriptor

We added ***Withdraw Response*** message to the triggers properties of the ***Withdraw Rejected*** plan.

Table 5.4 also shows that ***RulePJ5cc1*** rule identifies missing traceability relations between ***Request Approved***, ***Request Rejected***, ***Withdraw Money*** and ***Authorize Withdraw*** goals in Prometheus and some agents in JACK. We examined Prometheus models and found that we have not defined that Prometheus agents are supposed to achieve ***Request Approved***, ***Request Rejected***, ***Withdraw Money*** and ***Authorize Withdraw*** goals. We updated the model and defined that ***Request Approved***, ***Request Rejected***, and ***Withdraw Money*** goals in Prometheus are achieved by ***Atm*** agent and ***Authorize Withdraw*** is achieved by ***Bank*** agent.

Table 5.5 shows the results of our experiment after rectifying the documents and running the tool for the new versions of the models.

As shown in Table 5.5, the experiment achieved 100% of recall and 100% of precision, demonstrating an increase in the recall with respect to the initial results in the Table 5.3.

Number of relations identified by the engine |T|	64
Number of relations identified by the user |M|	64
Number of missing relations	0
Number of incorrect relations	0
Number of relations that intercept T and M	64
Recall	100%
Precision	100%

Table 5.5 Results of the experiments for the new models of the ATM case study

5.3 Air Traffic Control Environment

5.3.1 Overview of the Case Study

This subsection describes the development of a multi-agent system to implement the Air Traffic Control Environment used as a second case study in this thesis report. Air traffic congestion is a global issue and several air traffic management systems have already been built to alleviate this problem (Ljungberg, et al., 1992).

The air control environment consists of a system that implements arrival schedules at an airport. The main goal of an air control environment is to find the best landing time for an aircraft in order to alleviate congestion and its associated delays. A Feeder airport has the responsibility to process traffic of aircrafts. A Feeder airport contains information about all aircraft schedule arrivals that consists of the call sign (unique identifier of an aircraft used in the radio communications), booking time, ETA (Estimate Times of Arrival) to use for bookings, the arrival time at destination control area, and the ETA at control area entrance. The feeder airport waits until the booking time has passed and then sends the information to destination airport. A feeder aircraft receives update information about schedule changes such as a takeoff discard of an aircraft.

An aircraft sends a message to the airport when it enters a control area of the airport destination and waits until a runway has been allocated. To find the best landing time for an aircraft, the airport manager first queries all runway managers for the "best landing time" for an aircraft and then chooses one of them. After this, the airport manager notifies the decision to the runway manager and to the aircraft. In order to maximize the number of aircrafts that can land, faster aircraft that arrive later to the airport control area, push out earlier already assigned slower aircraft. A new bidding occurs to allocate a runway slot for the slower aircrafts. During the approaching to landing, the aircraft verifies continually if the runaway is still available for the aircraft to land until the landing time (ATL) has passed.

The next sections describe in detail the development of the Air Traffic Control Environment case study and its evaluation. The Air Traffic Control Environment case study used in our work was developed using *i** framework to model the organizational environment, Prometheus methodology to create the system specification, analysis and design models and JACK language to implement the system.

5.3.2 Artefacts

The Air Traffic Control Environment case study has been modelled using one Goal Overview diagram (see Figure 5.17) and four Capabilty Overview Diagrams (see Figure 5.19, Figure 5.20, Figure 5.21, and Figure 5.22). It is composed of four agents, namely Aircraft, Airport, Feeder, and Runway agents.

Table 5.6 shows the number of elements vs. types of elements in Prometheus used in the Air Traffic Control Environment case study.

Type of the element	Number of Elements
Agent	4
Capability	4
Goal	11
Message	4
Plan	10

Table 5.6 ATCE elements in Prometheus

Table 5.7 shows the number of elements vs. types of elements in *i** contained in the Air Traffic Control Environment case study.

Type of the element	Number of Elements
Actor	4
Goal dependency	2
Resource dependency	3
Task	10
Goal	2
Resource	3

Table 5.7 ATCE elements in *i**

Table 5.8 shows the number of elements vs. types of elements in JACK used in the Air Traffic Control Environment case study.

Type of the element	Number of Elements
Agent	4
BeliefSet	2
Capability	4
Event	4
Plan	10

Table 5.8 ATCE elements in JACK

5.3.3 Evaluation

The Air Traffic Control Environment case study was used to evaluate our work in order to:

- measure recall and precision of our tool;
- identify missing elements;
- show how the identified missing elements can support the development of the system by increasing its recall.

It is important highlight that this case study was used to demonstrate the performance of the tool in discovering and correcting defects in the models.

In order to demonstrate the above, we have (i) identified traceability relations between the models manually, (ii) identified traceability relations between the models using our tool, (iii) measure recall and precision by comparing the results in (i) and (ii), (iv) identified missing elements using our tool, (v) use the results in (iv) to manually to complete the models creating a new set of the models, (vi) use the tool to identify traceability relations and missing elements in the new models created in (iv), and (vii) measure the recall and precision of the results in (vi).

Our experiment has been conducted using 47 rules to identify traceability relations between Prometheus model and JACK code, 16 rules to identify missing elements between Prometheus elements and JACK code, 40 rules to identify traceability relations between *i** model and Prometheus model, and 18 rules to identify missing elements between *i** and Prometheus.

Table 5.9 shows the results of our experiments between Prometheus model and JACK code for (i), (ii) and (iii) activities.

Table 5.10 shows the results of our experiments between *i** model and Prometheus model for (i), (ii), and (iii) activities.

Number of relations identified by the engine |T|	125
Number of relations identified by the user |M|	153
Number of missing relations	34
Number of incorrect relations	6
Number of relations that intercept T and M	119

Recall	77.78%
Precision	95.2%

Table 5.9 Results of the experiments between Prometheus model and JACK code

Number of relations identified by the engine |T|	42
Number of relations identified by the user |M|	62
Number of missing relations	31
Number of incorrect relations	11
Number of relations that intercept T and M	31
Recall	50%
Precision	73.8%

Table 5.10 Results of the experiments between *i model and Prometheus model**

The missing traceability relations between Prometheus model and JACK code returned by the tool are shown in Table 5.11. The data of the traceability relations identified manually and by the tool is shown in the Appendix D.

Rule ID	JACK BeliefSet	Prometheus Data
RulePJ2cc1	LandingInfo	
RulePJ2cc1	RunwayInfo	
Rule ID	**Prometheus Goal**	**JACK Agent**
RulePJ5cc1	Request Slot	
RulePJ5cc1	Process Schedule for a Feeder	
RulePJ5cc1	Schedule Arrival for a Feeder	
RulePJ5cc1	Query Best Landing Time from All Runway Manager	
RulePJ5cc1	Assign Runway	
RulePJ5cc1	Find Best Land Time for an Aircraft	
RulePJ5cc1	Push Out	
RulePJ5cc1	Progresses an aircraft to Landing	
RulePJ5cc1	Initiate Aircraft Approach	
RulePJ5cc1	Assign Slot	
RulePJ5cc1	Landing	
Rule ID	**Prometheus Message**	**JACK Agent**
RulePJ12cc1	Traffic Event	
RulePJ12cc1	Enter Control Area	

Table 5.11 Missing relations between JACK code and Prometheus model

In order to show how missing elements identified by the tool can assist with the software development process (cases (iv) and (v)), missing relations given by the tool (see Table 5.11) fix inconsistencies (e.g. to fix discrepancies between names given by the elements). As shown in Table 5.11, there are missing relations between (a)JACK BeliefSet and Prometheus Data, (b)Prometheus Goal and JACK Agent, and (c) Prometheus Message and JACK Agent.

The rule RulePJ2cc1 identified that no overlaps traceability relations between LandingInfo and RunwayInfo beliefSet in JACK and Prometheus data has been found. Based on this information, we analysed the Prometheus model and then we concluded that included field properties for RunwayInfo (i.e. long ATL, String aircraft, long ETA, booking Boolean) and LandingInfo (i.e. String runway, long ATL) beliefsets in Prometheus model were missing to be defined.

The rule RulePJ5cc1 shows that no achieves traceability relations between Request Slot, Process Schedule for a Feeder, Schedule Arrival for a Feeder, Query Best Landing Time from All Runway Manager, Assign Runway, Find Best Land Time for an Aircraft, Push Out, Progresses an aircraft to Landing, Initiate Aircraft Approach, Assign Slot, and Landing goals and Prometheus have been found (see Table 5.11). The goals achieved by an agent in Prometheus have not been defined in the model.

Table 5.12 shows the results of our experiment after rectifying the models and running the tool for the new versions of the models. As shown in Table 5.12, the experiment achieved 100% of recall and 100% of precision, demonstrating an increase in the recall with respect to the initial results in Table 5.9.

Number of relations identified by the engine |T|	153
Number of relations identified by the user |M|	153
Number of missing relations	0
Number of incorrect relations	0
Number of relations that intercept T and M	153
Recall	100%
Precision	100%

Table 5.12 Results of the experiments for the new models of the ATCE case study

The missing relations between Prometheus model and *i** model returned by the tool are shown in Table 5.13. The data of the traceability relations identified manually and by the tool is shown in the Appendix D.

Rule ID	SD Goal	Goal
rule1cc	Allocate Runway Slot	
Rule ID	**SR Goal**	**---**
rule3cc	Allocate Runway Slot	
Rule ID	**SR Plan**	**Prometheus Goal || Prometheus Plan ||**

		Prometheus Role || Prometheus Action
rule4cc	Request Runway	
rule4cc	Respond Runway Request	
rule4cc	Follow Approach	
rule4cc	Request Booking	
rule4cc	TakeOff Discard	
rule4cc	TakeOff	
Rule ID	**Prometheus Goal**	**SD Task | SD Goal | SR Task | SD Goal**
rule4cc1	Request Slot	
rule4cc1	Schedule Arrival for a Feeder	
rule4cc1	Assign Runway	
rule4cc1	Push Out	
Rule ID	**Goal**	**Agent**
rule12		
Rule ID	**SD Resource**	**Percept | Message**
rule50cc	Slot Allocated	
rule50cc	ATL	
Rule ID	**SD Goal**	**Agent**
rule59cc1	Allocate Runway Slot	
Rule ID	**SR Goal**	**Agent**
rule59cc3	Allocate Runway Slot	
Rule ID	**SR Task**	**Agent**
rule59cc4	Request Runway	
rule59cc4	Respond Runway Request	
rule59cc4	Follow Approach	
rule59cc4	Request Booking	
rule59cc4	TakeOff Discard	
rule59cc4	TakeOff	
Rule ID	**SD Goal**	**Plan**
rule60cc1	Allocate Runway Slot	
Rule ID	**SR Goal**	**Plan**
rule60cc3	Allocate Runway Slot	
Rule ID	**SR Task**	**Plan**
rule60cc4	Request Runway	
rule60cc4	Respond Runway Request	
rule60cc4	Follow Approach	
rule60cc4	Request Booking	
rule60cc4	TakeOff Discard	
rule60cc4	TakeOff	
Rule ID	**SD Goal**	**Capability**
rule60cc1	Allocate Runway Slot	
rule60cc1	Find Best Landing Time for an Aircraft	
Rule ID	**SR Goal**	**Capability**
rule60cc3	Allocate Runway Slot	
rule60cc3	Find Best Landing Time for an Aircraft	
Rule ID	**SR Task**	**Capability**
rule60cc4	Request Runway	
rule60cc4	Respond Runway Request	
rule60cc4	Landing	

rule60cc4	Assign Slot	
rule60cc4	Initiate Approach	
rule60cc4	Follow Approach	
rule60cc4	Process Schedule for a Feeder	
rule60cc4	Request Booking	
rule60cc4	TakeOff Discard	
rule60cc4	TakeOff	

Table 5.13 Missing relations between *i and Prometheus model**

The completeness checking rule *rule1cc* shows that there is a missing traceability relation between Allocate Runway Slot SD Goal and a Prometheus goal. The rule rule3cc shows that there is a missing traceability relation between Allocate Runway Slot SR Goal and a Prometheus goal.

The rule rule4cc1 shows that no relation between Request Slot Prometheus goal and a SD Task, or SD Goal, or SR Task, or SR Goal has been found. The action taken to correct this discrepancy was to rename the name of Request Slot goal in the Prometheus model to Allocate Runway Slot. We changed the name of Request Slot to Allocate Runway Slot to fix the discrepancy between names and the traceability relation was identified.

The rule rule4cc shows that there are missing traceability relations between Request Runway, Respond Runway Request, Follow Approach, Request Booking, TakeOff Discard, TakeOff SR Tasks and Prometheus Goals. We added Request Booking and TakeOff Goal goals in the Prometheus to complete the model. After analysing the models, we identified that there is a missing relation between Follow Approach SR Task in *i** and Progress an aircraft to Landing in Prometheus model. To fix the discrepancy between the names given, we changed the name in the Prometheus model from Progress an aircraft to Landing to Follow Approach Goal.

The rule rule4cc1 shows that there are missing traceability relations between Request Slot, Schedule Arrival for a Feeder, Assign Runway, Push Out goals in Prometheus and SD Task, or SD Goal, or SR Task, or SR Goal. There is a discrepancy between the names given to the TakeOff Discard SR Task and Push Out goal in Prometheus. We changed the name of the SR Task from TakeOff Discard to Push Out in the *i** model. There is also a discrepancy between the names given to Respond Runway Request SR Task and Assign Runway in Prometheus. We changed the name of the Assign Runway goal in Prometheus to Respond Runway Request. No traceability relation was found between Schedule Arrival for a Feeder Prometheus goal and a SD Task, or SD Goal, or SR Task, or SR Goal. We decide to remove it from the Prometheus model. The Schedule Arrival for a Feeder is similar to the Process Schedule for a Feeder goal (another action could be to add a SR goal in the in *i** model that would have a means-end relations with the Process Schedule for a Feeder). The traceability relation between Request Runway SR Task in *i** and Query Best Landing Time from All Runway Manager was not identified by the tool. In order to fix this discrepancy between names we changed the name from Runway Request in the *i** model to Query Best Landing Time from All Runway Manager.

The rule rule50cc shows that no traceability relation was identified between Slot Allocated and ATL SD Resource in *i** and a Percept or a Message in Prometheus. We identified that the carried information ATL and Slot Allocated for the Aircraft Event message was missing. We fix the incompleteness adding ATL and Slot Allocated to the carried information property of Aircraft Event message.

The rule rule59cc1 shows that no traceability relation was identified between Allocate Runway Slot SD Goal and a Prometheus Agent and the rule rule59cc3 shows that no traceability relation was identified between Allocate Runway Slot SR Goal and a Prometheus Agent. We added to the list of goals achieved by the Runway Prometheus Agent the Allocate Runway Slot (named before by Request Slot).

The rule rule59cc4 shows that no traceability relation was identified between Request Runway, Respond Runway Request, Follow Approach, Request Booking, TakeOff Discard, and Take Off SR Tasks and agents in Prometheus. We added Query Best Landing Time from All Runway Manager (before named as Request Runway) to the list of goals achieved by Airport Prometheus Agent. No action was necessary for Request Runway Request and Follow Approach. The goal name changed from Assign Runway to Respond Runway Request and from Progresses an aircraft to Landing to Follow Approach resolved the incompleteness. We added Request Booking to the list of goals achieved by Aircraft Prometheus Agent and TakeOff and Push Out to the list of goals achieved by the Feeder Prometheus Agent.

Rule rule60cc1 shows that a relation is missing between Allocate Runway Slot SD goal and a Prometheus plan and rule rule60cc3 shows that a relation is missing between Allocate Runway

Slot SR goal and Prometheus plan. No action was necessary because the change of the Request Slot goal name to Allocate Runway Slot fixed the inconsistency.

Rule rule60cc4 shows that relations are missing between Request Runway, Respond Runway Request, Follow Approach, Request Booking, TakeOff Discard, and Take Off SR Tasks and plans in Prometheus. We added Query Best Landing Time from All Runway Manager (before named as Request Runway) to the list of goals achieved by Airport. No action was necessary for Request Runway Request and Follow Approach. The goal name changed from Assign Runway to Respond Runway Request and from Progresses an aircraft to Landing to Follow Approach resolved the incompleteness. We added Request Booking and TakeOff to the list of goals achieved by TakeOff Prometheus Agent. We added Push Out to the list of goals achieved by the Takeoff Discard Prometheus Plan.

Rule rule60cc1 shows that relations are missing between Allocate Runway Slot and Find Best Landing Time for an Aircraft SD Goal and a Prometheus Capability. The rule60cc3 shows that relations are missing between Allocate Runway Slot and Find Best Landing Time for an Aircraft SR Goal and a Prometheus Capability. We added Allocate Runway Slot goal to the list of goals achieved by Runway Assigning and Find Best Landing Time for an Aircraft to the list of goals achieved by Arrival Sequencing.

Rule rule60cc4 shows that relations are missing between Request Runway, Respond Runway Request, Landing, Assign Slot, Initiate Approach, Follow Approach, Process Schedule for a Feeder, Request Booking, TakeOff Discard, TakeOff SR Tasks in *i** and Prometheus Capability. We added Query Best Landing Time from All Runway Manager to the list of goals achieved by Arrival Sequencing capability. We added Respond Runway Request to the list of goals achieved by Runway Assigning. We added Landing, Assign Slot, Initiate Aircraft Approach, and Follow Approach to the list of goals achieved by Flying capability. We added Process Schedule for a Feeder, Request Booking, Push Out and Take Off Goal to the list of goals achieved by the Feeder capability.

Table 5.14 shows the results of our experiment after rectifying the models and running the tool for the new versions of the models. As shown in Table 5.14, the experiment achieved

78.82% of recall and 94.36% of precision, demonstrating an increase in the recall with respect to the initial results in Table 5.10.

We can observe also that the number identified by the user has changed from 62 to 71. This occurs because the *i** model and Prometheus model were modified by the creating of new elements and by consequence new relations were identified.

Number of relations identified by the engine |T|	85
Number of relations identified by the user |M|	71
Number of missing relations	4
Number of incorrect relations	18
Number of relations that intercept T and M	67
Recall	94.36%
Precision	78.82%

Table 5.14 Results of the experiments for the new models of the ATCE case study

5.4 Electronic Bookstore

5.4.1 Overview of the Case Study

This section describes the development of a multi-agent system to implement an Electronic Bookstore used as the third case study in this thesis report. The Electronic Bookstore allows the main tasks of buying and delivering books.

More specifically, the Electronic Book store allows a customer to browse the catalogue of books, buy books, search for books, and check the books delivery status. A customer starts a new order by selecting a book from the catalogue page or from the search result to a book. The customer specifies the quantity of books required and adds the book to the basket. The customer can continue to add more books to the order. He can also cancel the order entirely. Once the customer is satisfied with her (or his) selections, he (or she) checks out the order by entering a credit card number, name, billing address, and shipping address. The system sends a message to the credit card company to process the charge. If the credit card company approves the charge, the shipping clerk (person that pack an order) receives a shipping order and a label on their printer. The delivery company receives a message that an order will be ready for pickup. The shipping clerk packs the order and scans the barcode to indicate if the shipment is packed. The delivery company sends back a message to confirm when the order will be picked up. A shipment clerk records each shipment picked up by the delivery company.

The delivery company notifies the bookstore when the order has been delivered to the customer. The bookstore then e-mails the customer to inform that the order has been (or shortly will be) delivered.

The stock manager is a person that manages and controls the bookstore stock. The stock manager requests for new items from publishers, stores received books, purchases new books, updates catalogue of books, and controls inventory.

The sales manager is a person responsible for controlling the sale process and provides customers services before, during and after a purchase. The tasks realised by a sales manager is to keep prices competitive, reply to customer enquiries, and guarantee high level of customer satisfaction.

The Electronic Bookstore case study was used in two different experiments. Firstly, we used the tool to identify traceability relations between *i** and Prometheus model (Cysneiros, et al., 2007a) (Cysneiros, et al., 2007b). Secondly, we used the tool to identify traceability relations and to execute completeness checking between Prometheus model and JACK code. Although, we had used the same example for both experiments, the Prometheus model used by the second experiment is much more elaborated and reflects the implementation of the Electronic Bookstore in JACK.

The next sections describe in detail the development of the Electronic Bookstore case study and its evaluation. The Electronic Bookstore case study used in our work was developed using *i** framework to model the organizational environment, Prometheus methodology to create the system specification, analysis and design models and JACK language to implement the system.

5.4.2 Artefacts

In *i**, the Electronic Bookstore case study has been modelled using one Strategic Dependency model (see Figure 5.23) and one Strategic Rationale model (see Figure 5.24).

Table 5.15 shows the number of elements vs. types of elements in *i** contained in the Electronic Bookstore case study.

Type of the element	Number of Elements
Actor	7
Goal dependency	14
Resource dependency	4
Task	29
Goal	14
Resource	3

Table 5.15 EB elements in *i**

In Prometheus, the Electronic Bookstore has been modelled using one Goal Overview diagram (see Figure 5.25), one Role Model diagram (see Figure 5.26), one System Overview diagram (see Figure 5.1), one Use Case scenario (see Figure 5.27), five Agents Overview diagrams, and 39 Capability Overview diagrams.

Table 5.16 shows the number of elements vs. types of elements in Prometheus used in the Electronic Bookstore case study.

Type of the element	Number of Elements
Action	18
Agent	6
Capability	39
Data	8
Goal	37
Message	34
Percept	23
Plan	39
Scenario	25

Table 5.16 EB elements in Prometheus

Table 5.17 shows the number of elements vs. types of elements in JACK used in the Electronic Bookstore case study.

Type of the element	Number of Elements
Agent	6
BeliefSet	10
Event	37
Plan	52

Table 5.17 EB elements in Prometheus

The representation of Prometheus diagrams and JACK code in XML are presented in Appendix D.

5.4.3 Evaluation

The Electronic Bookstore case study was used to evaluate our work in order to:

- measure recall and precision of our tool;
- identify missing elements;
- show the use of the approach in a medium to large scale size example.

It is important highlight that this case study was used to demonstrate the performance of the tool in discovering trace relations.

In order to demonstrate the above, we have (i) identified traceability relations between the models manually, (ii) identified traceability relations between the models using our tool, (iii) measured recall and precision by comparing the results in (i) and (ii).

Our experiment has been conducted using 23 rules to identify traceability relations between *i** model and Prometheus model, 47 rules to identify traceability relations between Prometheus model and JACK code, and 16 rules to identify missing elements between Prometheus elements and JACK code.

Table 5.18 shows the results of our experiments between *i** and Prometheus models for (i), (ii), and (iii) activities.

Table 5.18 shows the results of the experiment. As shown in the table, the experiment achieved 86,36% of recall and 89,41% of precision.

Number of relations identified by the engine |T|	85
Number of relations identified by the user |M|	88
Number of missing relations	12
Number of incorrect relations	9
Number of relations that intercept T and M	76
Recall	86,36%
Precision	89,41%

Table 5.18 Evaluation Results

We have noticed that in our work the lower precision measurements occur when comparing goals and tasks in *i** SR and SD models with goals in Prometheus. This is due to the fact that this comparison takes into consideration the similarity between sub-elements of goals and tasks in *i** and sub-elements of goals in Prometheus, for the sub-elements in the various levels of the hierarchy. For instance, a sub-goal (or sub-task) E11 of a goal E1 in *i** may be incorrectly related to a goal E2 in Prometheus when the sub-elements of E2 match the sub-elements of E11. However, in reality, the similarity exists between E1 and E2, instead of E11 and E2.

As an example, consider *i** task "Place Order Online" in Figure 2 and Prometheus goal "Arrange delivery" in Figure 3. An overlaps traceability relation is created between these two elements given the fact that the sub-elements of "Arrange delivery" match sub-elements of "Place Order Online", although these elements do not refer to common aspects. We are currently changing some of our rules to avoid this situation.

Table 5.19 shows the results of our experiments between Prometheus model and JACK code for (i), (ii) and (iii) activities. The precision and recall calculated were 78,03% and 72,77%.

Number of relations identified by the engine |T|	651
Number of relations identified by the user |M|	698
Number of missing relations	190
Number of incorrect relations	60
Number of relations that intercept T and M	591
Recall	72,77 %
Precision	78,03 %

Table 5.19 Evaluation Results

The data of the traceability relations identified manually and by the tool is shown in Appendix D.

5.5 Discussion

The measurements for recall and precision from the above case studies are encouraging and comparable to recall and precision measurements achieved by other approaches that support automatic generation of traceability relations (Antoniol, et al., 2002), (Jirapanthong, et al., 2005), (Spanoudakis, et al., 2004), although these approaches deal with different types of models.

We recognized that the ideal it would have been used case studies that had been completely developed by third parties. The lack of these case studies make necessary to us develop some parts of the case studies. Although, we recognized the need to evaluate the approach with case studies developed completely by third parties that does not invalidate the results obtained. We developed the case studies using Prometheus methodology therefore we expected that other analysts would arrive to similar models if they had followed the methodology. It was also important to observe even when recall were low the missing element information helped to improve the recall. That makes us to conclude that even when the recall results are low the approach can helps to improve recall.

One question that could be raised is how the trace relations that are found manually can be used as a reference point to calculate recall and precision when these relations are derived from models inferred by the tool that are flawed and then stimulates changes to. The answer to that question is that the tool does not identify what is incorrect in the models. The tool identifies what is missing. The changes are to complete the models. We calculate recall and precision before this model are changed and then after these models have been changed. It works as two different experiments. Some of changes does not affect the number of relations identified manually since they are properties that are missing and the user had how to it. For instance, in Figure 5.6 the Process Withdraw plan receives the Withdraw Request message. The message is not defined as trigger to the Process Withdraw plan, but it is only message received by the plan the analyst can infer that message triggers the plan.

We also demonstrated for the ATM and ATCE case studies that the information about the missing elements was useful to fix discrepancies between names given to elements in the different abstract levels of documentation and to complete the models. The measurements for recall and precision of the amended models improved in comparison with the measurements for recall and precision for the original models.

Our experience has demonstrated that the implementation of the traceability and completeness checking rules is time-consuming. However, rules created for one case study for certain types of documents can be used for other cases studies for the same types of documents. Therefore, the created rules can be applied to other multi-agent systems developed using *i**, Prometheus, and JACK language.

The number of missing element rules implemented was not extensive. We believe that more rules could have been created to make the process of amending the model automatically.

When we create rules to identify traceability relations we have to define an *ideal* threshold to the degree of similarity between two elements required in order to create a traceability relation. This ideal threshold is not known and can interfere with the precision and recall measures. We did not make an exhaustive study to identify the ideal threshold for the various situations.

5.6 Threats of Validity

The threats for validity in the results of our evaluation are concerned with the fact that the analysis of the recall and precision of the work was performed by the same person that developed the traceability reference model for the documents of our concern, and the traceability and consistency checking rules. We discuss below the process in which we developed the traceability reference model and the rules, and how the above fact does not invalidate the results of the evaluation of the work.

In our work, the traceability reference model was created based on different types of traceability relations already proposed in the literature and the semantic of the different elements in the agent-oriented models used in our work. Therefore, the reference model was not adjusted to provide better recall and precision results, but was created independently of any foreseen results.

The traceability and consistency checking rules were created for the different types of traceability relations that we identified in the reference model taking into consideration the semantics of the various elements in the agent-oriented models and syntactic ways for describing the elements. Moreover, the specific models used to evaluate the work were chosen from different domains with different degrees of complexity in order to demonstrate a certain level of generalisation of the work when used in different examples. Furthermore, some of the models used in the case studies were provided by third parties such as the Prometheus model for Electronic Bookstore case study and the JACK code for the ATM and ATCE case studies. To construct the other models in the case study, we used the Prometheus methodology which is a well adopted methodology to develop agent-oriented systems, instead of creating the models in a way that will benefit the results of the evaluation.

Although the analysis of the results was executed by the same person that created the traceability and consistency checking rules, the analysis considered the results provided by the tool and the results of identifying the traceability relations manually. In the evaluation the assumption was that the traceability relations and completeness checking identified manually are the correct ones, and that the tool should try to identify, as much as possible, the same results as the ones manually identified. Therefore, it was necessary to have a person with good knowledge of the case studies and the traceability reference model to identify the traceability relations and missing elements manually. Moreover, during the evaluation of the work, the rules were not modified to improve recall and precision. Therefore, the person that created the rules did not interfere with the results of the experiments.

Another relevant point is concerned with the validity of scale of the work. More specifically, we are interested in evaluating if the success of the results of the work could be replicated in other case studies. We have evaluated the approach in three different cases chosen with distinct degrees of complexity and size in order to demonstrate that the approach scales and can used in different domains and case studies. The results of our evaluation were similar independently of the size, complexity, and type of the case studies.

In the work, we assume that analysts and programmers when creating software models in general, and agent-oriented models in particular, can miss the description of certain properties, elements, or relations between elements. To demonstrate that our work can support completeness checking we analyse the suggestions made by the tool about missing elements, modified the models used in the experiments with information about missing elements provided by our tool that were correct, and run the traceability generation again to identify relations involving the missing elements. We also use the modified documents to manually identify any new set of relations involving the new added elements and calculate recall and precision measures for the new sets of manually identified relations and the relations generated by the tool. Therefore, the new set of manually identified traceability relations that are used as reference to calculate recall and precision are based on analysis and verification of the suggestions returned by the tool.

5.7 Summary

This chapter have presented the results of evaluating our framework using three case studies namely (a) Automatic Teller Machine (ATM), (b) Air Traffic Control Environment (ACTE), and Electronic Bookstore (EB) systems. We describe the criteria for evaluation used to demonstrate the hypotheses of our research. For each case study, we presented a brief description, the created artefacts, and the results of the evaluation. Finally, we discussed the results obtained and compared these with other approaches.

Chapter 6 - Conclusion and Future Works

In this chapter, we present the conclusions, findings, and future works of this thesis report. In Section 6.1, we describe what was presented in this thesis. Section 6.2 shows how we achieve our hypotheses. Section 6.3 reviews the objectives of our work. In Section 6.4, we describe the contributions of this thesis. Section 6.5 shows possible directions for future works. Finally, Section 6.5 gives some final remarks.

6.1 Overall Conclusions

This thesis presented an approach to automatically identify traceability relations and missing elements in software models created during the development of multi-agent systems. In particular, we concentrate our work in software models generated when using *i** framework to represent the early requirements phase, Prometheus methodology to create analysis and design models, and JACK language to implement multi-agent systems.

Initially, in Chapter 2 we presented an overview of the software traceability research area. We defined what is software traceability, why traceability is important, and surveyed the main types of approaches to capture traceability relations semi- and automatically. We also discussed about the different classifications that have been proposed for different types of traceability relations and the various traceability reference models proposed in the literature. We reviewed some techniques to visualise, maintain, and use traceability relations.

In Chapter 3, we described the traceability reference model used in our work. First, we described the methodology that we based our work to develop multi-agent systems with its main software models and artefacts. Second, we defined different types of traceability relations associated with the artefacts of our concern and give some examples of these traceability relations. We proposed nine different types of traceability relations for artefacts composing the models *i** framework, Prometheus methodology, and the JACK language specifications. These traceability relation types are: satisfiability dependency, overlaps, evolution, implements, refinement, containment, similar, and different.

In Chapter 4, we presented our rule-based traceability framework to support (a) automatic generation of traceability relations of the types identified in Chapter 3 and (b) automatic identification of missing elements in the models of our concern. Our approach is rule-based and makes use of traceability and completeness checking rules specified in an extension of XQuery language. These rules support the identification of traceability relations and missing elements in the models generated during the development of multi-agent systems. We gave an overview of the traceability framework and explained the process on how to use the framework. We presented the traceability and completeness checking rules and gave some examples of these rules. We specified a list of functions created in Java to extend XQuery in order to specify traceability and completeness checking rules. We also described the prototype tool that we have developed to support the traceability framework.

In Chapter 5, we evaluated our traceability framework in three different case studies, namely (a) Automatic Teller Machine (ATM), (b) Air Traffic Control Environment (ATCE), and (c) Electronic Bookstore (EB) systems. We evaluated our work in terms of recall and precision measurements. We also evaluated how the identification of missing elements by our framework can increase precision of the traceability relations after amending the models with the information identified about the missing elements. The recall and precision measurements in our evaluation were encouraging and comparable to recall and precision measurements achieved by other approaches that support automatic generation of traceability relations (Antoniol, et al., 2002), (Jirapanthong, et al., 2005), (Spanoudakis, et al., 2004), although these approaches deal with different types of models and case studies.

Table 6.1 shows a summary of the results of recall and precision of our experiments. As shown in the table, recall measurements range between 50% – 100 % and precision measurements range between 73.80% – 100%. The headings of the columns in the first row of the table represent each of the different case studies and their respective executions, as explained below.

	ATM_1	ATM_2	$ATCE_1$	$ATCE_2$	$ATCE_3$	$ATCE_4$	EB_1	EB_2
Recall	50%	100%	77.78%	100%	50%	94.36%	86.36%	72.77%
Precision	100%	100%	95.20%	100%	73.80%	78.82%	89.41%	78.03%

Table 6.1 – Results of the experiments

ATM_1 – results of the first round of experiments between Prometheus model and JACK code for the Automated Teller Machine case study.

ATM_2 – results of the second round of experiments between Prometheus model and JACK code for the Automated Teller Machine case study, after amending the models based on the identified missing elements in the first round of the experiments (ATM_1).

$ATCE_1$ – results of the first round of experiments between Prometheus model and JACK code for the Air Traffic Control Environment case study.

$ATCE_2$ – results of the second round of experiments between Prometheus model and JACK code for the Air Traffic Control Environment case study, after amending the models based on the identified missing elements in the first round of the experiments ($ATCE_1$).

$ATCE_3$– results of the first round of experiments between *i** and Prometheus models for the Air Traffic Control Environment case study.

$ATCE_4$ – results of the second round of experiments between *i** and Prometheus models for the Air Traffic Control Environment case study, after amending the models based on the identified missing elements in the first round of the experiments ($ATCE_3$).

EB_1 – results of the first round of experiments between *i** and Prometheus models for the Electronic Bookstore case study.

EB_2 – results of the first round of experiments between Prometheus model and JACK code for the Electronic Bookstore case study.

The results of recall and precision measurements in our evaluation are encouraging and comparable in terms of absolute values with the recall and precision measurements achieved by other approaches that support automatic generation of traceability relations (Marcus, et al., 2003), (Zisman, et al., 2002), (Zisman, et al., 2003), (Spanoudakis, et al., 2004). It has to be noted, however, that these other approaches consider different types of documents and different case studies.

For example, in (Zisman, et al., 2002), (Zisman, et al., 2003), (Spanoudakis, et al., 2004), the authors evaluated their rule-based approach for automatic generation of traceability relations in a case study of a family of software-intensive TV systems and in a case study for a university course management system composed of requirements documents, use case models, and analysis object models represented as a UML class diagram. In this work, the results of recall and precision measurements range between 50% and 95% for a family of software-intensive TV systems and for a university course management system case studies used to evaluate a rule based approach proposed in (Zisman, et al., 2002), (Zisman, et al., 2003), (Spanoudakis, et al., 2004) . Traceability relations identified were between requirements statements, use cases, and analysis object model. Recall measurements range between 65.4% – 92.0% and precision measurements range between 81.0 – 90.5% (See Table 6.2) for a mobile phone product line system case study used to evaluate a rule base approach proposed in (Jirapanthong, et al., 2005). Traceability relations were identified between feature, subsystem, process, use cases, class diagram, statechart, and sequence diagrams used to represent a product line system.

	Case 1	*Case 2*	*Case 3*	*Case 4*	*Case 5*	*Average*
Precision	90.5	89.8	81.6	81.0	83.4	**85.3**
Recall	92.0	90.6	85.4	65.4	83.4	**83.3**

Table 6.2 Results of the experiments

It is difficult to compare results of precision and recall measurements obtained from experiments using information retrieval techniques to identify traceability relations with recall and precision measurements from experiments using rule base approaches, since precision and recall values depend on the use of threshold or cut point values.

To select candidate traceability relations, these approaches consider traceability relations between artefacts that have a degree of similarity greater than a pre-defined threshold or make use of cut points of ranked traceability relations in which only the n top best ranked relations are selected. However, the specification of threshold values is difficult, because recall improves with the increment of the threshold, but precision deteriorates with the decrement of the threshold. Similarly, the decision of what should be the cut point is also difficult, since recall improves with the increment of the cut point, but precision deteriorates with the increment of the cut point. In general, an initial threshold (or cut point) value is selected and this value is modified (incremented or decremented) until an ideal value is found.

Table 6.3 shows recall and precision measurements when the threshold is equal to 0.60%, equal to 0.65%, and 0.70% for an experiment with a case study of the LEDA system (Library of Efficient Data types and Algorithms) used to evaluate an information retrieval approach applying Latent Semantic Indexing method (LSI) (Marcus, et al., 2003). Recall is equal to 42.98% and precision is equal to 71.01% when the threshold is equal to 0.60%. Recall is equal to 53.97% and precision is equal to 59.65% when the threshold is equal to 0.65%. Recall is equal to 42.63% and precision is equal to 71.05% when the threshold is equal to 0.70%. Table 6.4 show recall and precision measurements when a cut point is used to select the candidate relations. Recall is equal to 59.65% and precision is equal to 77.27%, is equal to 53.98% and precision is equal to 83.33%, 40.53% and precision is equal to 93.86%, is equal to 30.97% and precision is equal to 95.61% when the cut point is equal to 1, 2, 3 and 4, respectively.

Threshold	*Precision*	*Recall*
0.60	*42.98%*	*71.01%*
0.65	*53.97%*	*59.65%*
0.70	*71.05%*	*42.63%*

Table 6.3 – Results of LEDA case study using threshold

Cut Point	*Precision*	*Recall*
1	*77.27%*	*59.65%*
2	*53.98%*	*83.33%*
3	*40.53%*	*93.86%*
4	*30.97%*	*95.61%*

Table 6.4 Results of the experiments

As shown in the table 6.1, our framework achieves positive results, with 100% of recall and precision in some specific cases. Moreover, apart from automatic generation of traceability relations, our framework also allows automatic identification of missing elements between the models (completeness checking) and, therefore, provides a better support for the development of multi-agent systems. The results of our experiments have demonstrated our hypotheses (see Chapter 6.2).

6.2 Hypotheses

In this section, we review the hypotheses of our work described in Chapter 1 and show how we achieved our initial hypotheses. Here, we present the hypotheses again for your reference.

Hypothesis: *We can use rules to identify traceability relations between software artefacts created during the development of multi-agent systems using a model driven approach.*

We demonstrated the hypothesis using recall and precision measurements. A possible threat of validity is that in our experiments the same person that created the diagrams is the same person that created the rules. We believe that our findings would be similar even if the person that created the rules was different from the person that created the diagrams. This is supported by the fact that the rules are created based on the Prometheus methodology and the person that created the diagrams follows the Prometheus methodology. Another person following Prometheus methodology would follow the same guidelines and arrive to similar results.

We have developed a framework and a prototype tool based on the use of several rules to support the automatic generation of nine different types of traceability relations in software models created when using the *i** framework, Prometheus methodology, and JACK language. The evaluation of our work showed high level results for recall (50%-100%) and precision (73.8%-100%) measurements.

As defined in the Section 5.1, precision is the percentage of correct relations identified by the tool and recall is the percentage of relevant relations identified by the tool. Therefore, precision and recall measures were used to demonstrate that the tool can identify traceability relations between software artefacts created during the development of multi-agent systems modelled using *i** framework, Prometheus methodology, and JACK language.

Table 6.1 show recall and precision measurement for Automatic Teller Machine (ATM), Air Traffic Control Environment (ATM), and Electronic Bookstore (EB) case studies. The lowest value of recall is 50% for ATM_1 and $ATCE_3$ that represent recall measurements before to use the missing elements information to complete the models and to fix discrepancies between names. All the other recall measurements are above 70%. After to use the missing elements information to complete the models and to fix the discrepancies of names for the ATM and ATCE case studies, the lowest recall measurement value is 72.77% for EB_2 where the information about missing elements were not used to complete the models and to fix discrepancies between names.

The lowest value of precision is 73.80% for $ATCE_3$ case study. All the other precision measurements are above 78%. After to use the missing elements information to complete the models and to fix the discrepancies between names for the case studies, the lowest precision measurement value is 78.03% for EB_2 case study.

As discussed, measures of recall and precision for Automatic Teller Machine (ATM), Air Traffic Control Environment (ATM), and Electronic Bookstore (EB) case studies were encouraging and comparable to recall and precision measurements achieved by other approaches that support automatic generation of traceability relations (Antoniol, et al., 2002), (Jirapanthong, et al., 2005), (Spanoudakis, et al., 2004).

Hypothesis: *We can use rules to identify missing elements between software artefacts created during the development of multi-agent systems using a model driven approach.*

We created several rules to support the automatic identification of missing elements in software models created when using the *i** framework, Prometheus methodology, and JACK language. The ATM and ATCE case studies were used to demonstrate this hypothesis. The table 5.4 shows a list of missing elements to the ATM case study and tables 5.11 and 5.13 show a list of missing elements to the ATCE case study.

Hypothesis: *We can use the information about missing elements to fix discrepancies between names given to elements in the different abstract levels of documentation and to improve completeness between software artefacts created during the development of multi-agent systems using a model driven approach.*

Our framework and prototype tool adopts a process in which information about the identified missing elements is used to amend the models and the new versions of the models are used to generate new traceability relations.

For instance in the ATCE case study, the rule rule4cc shows that there is a missing traceability relations between Follow Approach and a Prometheus Goal. After analysing the models, we identified that there is a missing relation between Follow Approach SR Task in *i** and Progress an aircraft to Landing in Prometheus model. To fix the discrepancy between the names given, we changed the name in the Prometheus model from Progress an aircraft to Landing to Follow Approach Goal.

Hypothesis: *We can use the information about missing elements to improve the number of traceability relations identified by our approach.*

The results of generating traceability relations in the amended models, based on the identified missing elements by our framework and tool, show an increase in the precision and recall measurements in our case studies. Table 6.5, Table 6.6, and Table 6.7 show that the number of traceability relations increase when the information about the missing elements is used to amend the models and to fix discrepancies in names of the artefacts, for our three case studies. More specifically, Table 6.5 shows that the number of traceability relations increases from 31 to 64 with recall measurement increasing from 50% to 100%, for the Automatic Teller Machine case study. In this case there was no change of precision measurements given an initial 100% measurement.

	ATM_1	ATM_2
Recall	50%	100%
Precision	100%	100%
Relations	31	64

Table 6.5 Number of traceability relations identified for the ATM case study

Table 6.6 shows that the number of traceability relations increases from 125 to 153 with recall measurements increasing from 77.78% to 100% and precision measurements increasing from 95.20 to 100%, for the Airtrafic Control Environment case study.

	$ATCE_1$	$ATCE_2$
Recall	77.78%	100%
Precision	95.20%	100%
Relations	125	153

Table 6.6 Number of traceability relations identified for ATCE case study

Table 6.7 shows that the number of traceability relations increases from 42 to 85 with recall measurement increasing from 50% to 94.36% and precision measurement increasing from 73.8.0 to 78.82%, for the Eletronic Bookstore case study.

	$ATCE_3$	*$ATCE_4$*
Recall	50%	94.36%
Precision	73.80%	78.82%
Relations	42	85

Table 6.7 Number of traceability relations identified for the ATCE case study

6.3 Objectives

In this section, we review the objectives of our work described in Chapter 1. Here, we present the objectives again for your reference.

Objective 1: To define traceability relation types

We defined nine different types of traceability relation for artefacts in the software models created when using *i** framework, Prometheus methodology, and JACK code specifications. These traceability relation types are: overlaps, contributes, uses, creates, achieves, depends on, composed of, send and receives.

Objective 2: To create a reference model that defines traceability relations between artefacts in *i** and Prometheus and between artefacts in Prometheus and JACK code.

We created a traceability reference model for traceability relations between *i** and Prometheus artefacts and a traceability reference model for traceability relations between Prometheus and JACK code artefacts.

Objective 3: To create a set of rules to identify missing elements and traceability relations between *i** and Prometheus artefacts

Objective 4: To create a set of rules to identify missing elements and traceability relations between *i** and Prometheus artefacts and between Prometheus and JACK artefacts.

We created a set of traceability and completeness checking rules to identify traceability relations and missing elements between (a) *i** and Prometheus artefacts and (b) Prometheus and JACK code artefacts.

Objective 5: To develop a prototype tool to identify missing elements and that automatically generates traceability relations between *i** and Prometheus and between Prometheus and JACK code models. A prototype tool has been implemented.

Objective 6: To evaluate the approach in several case studies

We evaluated the framework and prototype tool in three different small to large-size case studies, namely (a) Automatic Teller Machine (ATM), (b) Air Traffic Control Environment (ATCE), and (c) Electronic Bookstore (EB) systems. Each of these case studies demonstrated a different aspect of our work.

6.4 Contributions

The work described in this thesis contributes in general to the broader area of agent oriented software engineering in three main aspects. More specifically, it provides a traceability reference model for documents generated during the development of multi-agent systems, it supports the automatic generation of nine different types of traceability relations between the various artefacts created during the development of multi-agent systems, and it supports completeness checking of these various artefacts.

As discussed in (Ramesh, et al., 2001), (Dick, 2002), (Spanoudakis, et al., 2005), a traceability reference model in which the semantic of traceability relations are specified helps with the execution of richer analysis about traceability relations and assists with certain software engineering activities such as maintenance, impact analysis, and reuse. To the best of our knowledge, no traceability reference model that covers all the stages of the development life-cycle of multi-agent systems based on the use of *i** framework, Prometheus methodology, and JACK code has been proposed.

Pinto et al. proposed a traceability reference model for the Tropos methodology (Pinto, et al., 2005). This reference model offers a higher level of granularity for the types of traceability relations when compared to our traceability reference model.

We developed a rule-based approach to automatically generate traceability relations and identify missing elements in software artefacts produced during the development of multi-agent systems using *i** framework, Prometheus methodology, and JACK code.

The developed work uses an attribute to indicate the level of confidence in a generated traceability relation (*degreeOfCompleteness*) between two artefacts. This information can be used to support analysis of the related artefacts.

Our approach also executes completeness checking. The approach executes completeness checking in two different ways. The first method is to identify what elements are missing to be created in the target model from the source model perspective, or vice-versa. The second method is when traceability relations are created between two elements and some incompleteness is found because some discrepancies of names, properties, sub-elements or relations are missing.

6.5 Future Work

The work presented in this thesis opens a number of possible directions for further investigations. We describe in this section future work that needs to be developed in order to improve the approach and increase its benefits.

- Traceability visualisation tool
- GUI tool to support completeness checking
- Evolution of traceability relations
- Translator between JACK code in XML
- Support for other modelling and programming languages
- Graphical editor for traceability rules tool to support customisation of rules
- Graphical tool to support the creation of rules

Traceability visualisation tool

In our research, we only address the problem of identifying traceability relations. We believe that it would be beneficial to extend our framework and tool to support visualisation of traceability relations. Our experience has shown that a large number of traceability relations can be generated for the various models and it will be important to have a better way of visualising and manipulating these relations. The current version of our tool allows the creation of HTML reports of traceability relations to be displayed on a browser, but this is very limited. Marcus et al. discuss and propose in (Marcus, et al., 2003) a set of high level requirements for a tool that support visualisation of traceability relations such as i) allow the user to browse the traceability relations through various types of user interactions; ii) allow the user to add, remove, and modify properties of existing traceability relation and their related artefacts; iii) integration with tools used to develop, test and maintain the system; iv) capture and maintain browsing history for traceability relations; v) provide comprehensive configuration management and change tracking facilities; vi) support various data representation formats; vii) support user querying and filtering of the traceability relations; viii) offer flexible and user customizable view of the traceability data; ix) provide tools to analyse and summarize the data on the traceability process and relations.

GUI tool to support for completeness checking activity

In our approach, we only point out the missing information and discrepancies between names of the elements. The action to complete the models or to fix the discrepancies has to be done manually. We believe that a tool could be developed to support in the activity of complete the models and to fix the discrepancies suggesting actions or even executing actions semi-automatically or automatically.

Evolution of traceability relations

We did not address the problem of evolution of the traceability relations in our research. There are several works that address this problem (Cleland-Huang, et al., 2003), (Sharif, et al., 2007), (Murta, et al., 2006a), (Murta, et al., 2006b). One solution it would be to investigate if our approach could be integrated with one of those approaches.

Translator between JACK code in XML

Some work has been done in the direction to create a translator to transform JACK code in plain/text into XML format. The implementation of the translator has not been completed. A translator to transform JACK code in plain/text into XML format has been implemented as part of a final year project by an undergraduate student. The tool does not cover all the elements in JACK language that is required by our approach and the XML does not fit completely to the requirements of our approach. We could modify the tool created to meet the requirements of our approach.

Support for other modelling and programming languages

One of the challenges in the development of multi-agent systems is to provide support to the vast number of methodologies, platforms, tools, and languages available to develop multi-agent systems. Our approach supports models created when using *i** framework, Prometheus methodology, and JACK code. In particular, the traceability and completeness checking rules used in our work can supports *i** models created using TAOME tool (TAOME4E, 2008) and Prometheus model created using PDT tool (PDT, 2010). Moreover, the use of an XML and XQuery-based approach has shown that it is possible to deal with heterogeneous models generated by different tools and with different semantics. We believe that our approach is general enough to be adapted for other types of artefacts that may be generated when using other methodologies, frameworks, or platforms for the development of multi-agent systems.

Graphical editor for tool to support customisation of rules

We believe that different organizations, projects and stakeholders have different needs in terms of traceability. In our approach, we created a traceability reference model that can be customised to specific needs. The tool could allow the user to select the rules to use in a project and provide mechanisms to select what rules to use based on specific criterion such as type of relations, or artefacts. At the present, no support is provided to the selection of the rules. The user has to manually create a file with all the rules that are executed by the tool.

Graphical tool to support the creation of rules

Our approach relies on the use of traceability and completeness checking rules specified in an extension of XQuery. We agree that the creation of these rules is not a straight forward activity and requires knowledge of XQuery, and use of some extra functions. To make the approach more flexible, it would be interesting to provide an editor to support the creation of traceability and completeness checking rules using graphical approaches such as the functionality provided by Microsoft Access to support the creation of queries graphically (Microsoft Corp., 2010). We provide a template for the rules that can be used as a basis of a rule tool editor.

6.5 Final Remarks

During the development of multi-agent systems a large number of artefacts are produced and the relation between these artefacts is complex and difficult to identify manually. This thesis presented an approach to support automatic generation of traceability relations and to support completeness checking between software artefacts created during the development of multi-agent systems using the BDI architecture. Chapter 2 we presented an overview of the software traceability research area. In Chapter 3, we described the traceability reference model used in our work. Chapter 4 described our rule-based traceability framework to support (a) automatic generation of traceability relations of the types identified in Chapter 3 and (b) automatic identification of missing elements in the models of our concern based. In Chapter 5, we evaluated our traceability framework in three different case studies, namely (a) Automatic Teller Machine (ATM), (b) Air Traffic Control Environment (ATCE), and (c) Electronic Bookstore (EB) systems.

Bibliography

Agent Oriented Software Limited. (2010). *JACK white paper*. Retrieved December, 2010, from http://www.aosgrp.com/downloads/JACK_WhitePaper_UKAUS.pdf

Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., & Shaham-Gafni, Y. (2006). Model traceability. *IBM Systems Journal*, *45*(3), 515-526.

Almeida, J., Iacob, M., & Eck, P. (2007). Requirements traceability in model-driven development: Applying model and transformation conformance. *Information Systems Frontiers*, *9*(4), 327-342.

Alves-Foss, J., Conte de Leon, D., & Oman, P. (2002). Experiments in the use of XML to enhance traceability between object-oriented design specifications and source code. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02) – Volume 9*.

Amazon.com. (2010). *Help*. Retrieved December, 2010, from http://www.amazon.com/gp/help/customer/display.html

Antoniol, G., Canfora, G., & de Lucia, A. (1999). Maintaining traceability during object-oriented software evolution: A case study. *Proceedings of the IEEE International Conference on Software Maintenance*, 211-219.

Antoniol, G., Caprile, B., Potrich, A., & Tonella, P. (2001) Design-code Traceability Recovery: Selecting the basic linkage properties. *Science of Computer Programming*, *40*, 213-234.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation, *IEEE Transactions on Software Engineering*, *28* (10), 970-983.

Asuncion, H., François, F., & Taylor, R. (2007). An end-to-end industrial software traceability tool. *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia.

Berenbach, B. (2007). Managing traceability with software models. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Bordini, R., Hubner, J., & Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix & A. El Fallah Seghrouchni (Eds.), *Multi-agent programming: Languages, platforms and applications*. (pp. 3-37). New York: Springer.

Borland. (2010). *CaliberRM*. Retrieved December, 2010, from http://www.borland.com/us/products/caliber/

Bratman, M., Israel, D., & Pollack, M. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence, 4*(3), 349-355.

Bratman, M. (1999). *Intentions, plans and practical reason*. Cambridge University Press.

Burge, J. & Brown, D. (2007). Supporting requirements traceability with rationale. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Busetta, P., Rönnquist, R., Hodgson, A., & Lucas, A. (1999). JACK Intelligent Agents - components for intelligent agents in Java. *AgentLink Newsletter*, *2*, 2-5.

Carnegie Mellon. (2010). *Capability Maturity Model Integration*. Retrieved December, 2010, from http://www.sei.cmu.edu/cmmi/

Castro J., Kolp M. & Mylopoulos J. (2002). Towards requirements-driven information systems engineering: The Tropos project. *Information System, 27*(6), 365-389.

Castro-Herrera, C. & Cleland-Huang, J. (2007). Towards a unified process for automated traceability. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Cleland-Huang, J., Chang, C., & Christensen, M. (2003). Event-based traceability for managing evolutionary change. *IEEE Trans. Software Eng., 29*(9), 796-810.

Cleland-Huang, J., Dekhtyar, A., & Hayes, J. (2007). Problem statements and grand challenges. (COET-GCT-06-01-0.9). Center of Excellence for Traceability.

Cockburn, A. *Writing effective use cases* (2000). Addison-Wesley Professional.

Cysneiros, G., Zisman, A., & Spanoudakis, G. (2003). A traceability approach for *i** and UML models. *Proceedings of 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - ICSE 2003*, Portland, USA.

Cysneiros, G. & Zisman, A. (2007). Traceability for agent-oriented design models and code. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Cysneiros, G. & Zisman, A. (2007). Tracing agent-oriented systems. *Proceedings of the The Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2007)*, Boston, Massachusetts, USA, 552-558.

Cysneiros, G., & Zisman, A. (2008). Traceability and completeness checking for agent-oriented systems. *Proceedings of the 2008 ACM Symposium on Applied Computing*, Fortaleza, Ceara, Brazil. 71-77.

Dagenais, B., Breu, S., Warr, F. W., & Robillard, M. P. (2007). Inferring structural patterns for concern traceability in evolving software. *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA. 254-263.

Davis A. (1990). The analysis and specification of systems and software requirements. *Systems and Software Requirements Engineering*. IEEE Computer Society Press, 119-144.

De Lucia, A., Fasano, F., Oliveto, R., & Tortora, G. (2004). Enhancing an artefact management system with traceability recovery features. *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 306-315.

Lucia, A. D., Fasano, F., Oliveto, R., & Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans.Softw.Eng.Methodol., 16*(4).

De Lucia, A., Oliveto, R., & Tortora, G. (2008). Adams re-trace: Traceability link recovery via latent semantic indexing. *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany. 839-842.

DeLoach, S. (2001). Analysis and design using MaSE and agentTool. *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference.*

Dick, J. (2002). Rich Traceability. Proceedings of the 1st International Workshop on Traceability for Emerging forms of Software Engineering . Edinburgh, UK

d'Inverno, M., Luck, M., Georgeff, M., Kinny, D., & Wooldridge, M. (2004). The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Journal of Autonomous Agents and Multi-Agent Systems*, 5-53.

Duan, C. & Cleland-Huang, J. (2007). Clustering support for automated tracing. *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA. 244-253.

Egyed, A. (2003). A scenario-driven approach to trace dependency analysis. *IEEE Transactions on* Software Engeneering, *29*(2), 116.

Egyed, A. & Grünbacher, P. (2005). Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge* Engineering, 15, 783-810.

Fiutem, R. & Antoniol, G. (1998). Identifying design-code inconsistencies in object oriented software: A case study. *Proceedings of the International Conference on Software Maintenance*, 94-102.

Fletcher, J. & Cleland-Huang, J. (2007). Utilizing softgoal traceability patterns to monitor design goals. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Frisch, A. (2002). *Essential System Administration* (3rd ed.). O' Reilly.

Genesereth, M. R. & Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Goknil, A., Kurtev I. & Berg, K. (2008). Change impact analysis based on formalization of trace relations for requirements. *Proceedings of the ECMDA Traceability Workshop*. Berlin, Germany.

Gotel, O. & Finkelstein, A. (1994). An analysis of the requirments traceability problem. *Proceedings of the 1st International Conference on Requirements Engineering*. 94-101.

Gotel, O. & Morris, S. (2007). From Farm to Fork or a Bite of the Unknown: Learning from the Food Industry. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Gotel, O. (2008). yTraceability – Putting the 'y' first. *Requirements Quarterly: The Newsletter of the Requirements Engineering Specialist Group of the British Computer Society. RQ50.*

Grechanik, M., McKinley, K. S., & Perry, D. E. (2007). Recovering and using use-case-diagram-to-source-code traceability links. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia. 95-104.

Han, J. (2001). TRAM: A tool for requirements and architecture management. *Proceedings of the 24th Australasian Computer Science Conference.* 60-68.

Hayes, J. H., Dekhtyar, A., Sundaram, S. K., & Howard, S. (2004). Helping analysts trace requirements: An objective look. *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE)*, 249-259.

Hayes, J. H., Dekhtyar, A., Sundaram, S. K., Holbrook, E. A., Vadlamudi, S., & April, A. (2007). REquirements TRacing on target (RETRO): Improving software maintenance through traceability recovery. *ISSE, 3*(3), 193-202.

Hollings, D., Kieran, S., & Kelleher, J. (2005). *Representing Requirements Traceability Using XML Topic Maps.* (Report CS05-20-00). Department of Computer, University of Cape Town Science.

Howden, N., Ronnquist, R., Hodgson, A., & Lucas, A. (2001). JACK intelligent agents - summary of an agent infrastructure. *Proceedings of the 5th ACM International Conference on Autonomous Agents*.

Huber, M. J. (1999). JAM: A BDI-theoretic mobile agent architecture. *Proceedings of the Third Annual Conference on Autonomous Agents*, Seattle, Washington, United States. 236-243.

Huget, M. P., Odell, J., Nodine, M. M., Cranefield, S., Levy, R., & Padgham, L. (2003). *Fipa modeling: Interaction diagrams*. (FIPA Working Draft, version 2003-07-02). Foundation for Intelligent Physical Agent.

IBM Rational. (2010). *Rational DOORS*. Retrieved December, 2010, from http://www-01.ibm.com/software/awdtools/doors/

IBM Rational. (2010). *Rational RequisitePro*. Retrieved December, 2010, from http://www-01.ibm.com/software/awdtools/reqpro/

Ingrand, F. F., Georgeff, M. P., & Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert, 7*(6), 34-44.

ISO. (2010). *ISO Standards*. Retrieved December, 2010, from http://www.iso.org/iso/iso_catalogue.htm

Jirapanthong, W. & Zisman, A. (2005). Supporting product line development through traceability. *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, 506-514.

Jirapanthong, W. & Zisman, A. (2009). XTraQue: Traceability for product line systems. *Software and Systems Modeling, 8*(1), 117-144.

Kagdi, H. & Maletic J. (2007). Software repositories: A source for traceability links. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Kritzinger P. & Krüger H. (2008). *Software Traceability using Latent Semantic Analysis and Relevance Feedback*. (Report CS05-20-00). Department of Computer, University of Cape Town Science.

Kurtev, I., Dee, M., Göknil, A., & Berg, K. G. v. d. (2007). Traceability-based change management in operational mappings. *Proceedings of the ECMDA Traceability Workshop 2007*, 57-67.

Lindvall, M. & Sandahl, K. (1996). Practical implications of traceability. *Software Practice and Experience, 26*(10), 1161-1180.

Liu, D., Marcus, A., Poshyvanyk, D., & Rajlich, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA. 234-243.

Ljungberg, M. & Lucas, A. (1992). The OASIS air traffic management system. *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*.

Lormans, M. & Van Deursen, A. (2006). Can LSI help reconstructing requirements traceability in design and test? *Proceedings of the Conference on Software Maintenance and Reengineering*, 47-56.

Luck M., Ashri R., & d'Inverno M. (2004). *Agent-Based Software Development*. Artech House Publishers.

Maletic, J. I., Munson, E. V., Marcus, A., & Nguyen, T. N. (2003). Using a hypertext model for traceability link conformance analysis. *Proceedings of the 2nd Int. Workshop on Traceability in Emerging Forms of Software Engineering*, 47-54.

Marcus, A., Maletic, J., & Sergeyev, A. (2005). Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering,15* (5), 811-836.

Marcus, A. & Maletic, J. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon. 125-135.

Marcus, A., Maletic, J. I., & Sergeyev, A. (2005). Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering, 15*(4), 811-836.

Mohan, K., Xu P., & Ramesh. (2008). B. Improving the change management process: Traceability meets configuration management. *Communications of the ACM, 51*(5): 59-64.

Murta, L. G. P., van der Hoek, A., & Werner, C. M. L. (2006). ArchTrace: Policy-based support for managing evolving architecture-to-implementation traceability links. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 135-144.

Murta L., van der Hoek A. & Werner C. (2006). ArchTrace: A tool for keeping in sync architecture and its implementation. *Proceedings of the Brazilian Symposium on Software Engineering (SBES).*

OMG. (2010). *UML 2.2 Specification*. (Formal/2009-02-02). Retrieved December, 2010, from http://www.omg.org/spec/UML/2.3/

Padgham L. & Winikoff M. (2004). *Developing intelligent agent systems: A practical guide*. John Wiley and Sons.

Padgham, L., Thangarajah, J., & Winikoff, M. (2005). Tool support for agent development using the prometheus methodology. *Proceedings of the Fifth International Conference on Quality Software*, 383-388.

Padgham, L., Thangarajah, J., & Winikoff, M. (2007). AUML protocols and code generation in the prometheus design tool. *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, Honolulu, Hawaii.

PDT. (2010). *Prometheus Design Tool (PDT)*. Retrieved December, 2010, from http://www.cs.rmit.edu.au/agents/docs.shtml

Perini, A., & Susi, A. (2005). Automating model transformations in agent-oriented modelling. *Proceedings of of 6th International Workshop AOSE.*

Pinheiro, F. A. C. & Goguen, J. A. (1996). An object-oriented tool for tracing requirements. *IEEE Software, 13*(2), 52-64.

Pinto, R., Silva, C. & Castro J. (2005). Support for requirement traceability: The Tropos case. *Proceedings of the 19th Brazilian Symposium on Software Engineering*.

Pohl, K. (1996). Process-centered requirements engineering. John Wiley & Sons, Inc.

Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix & A. El Fallah Seghrouchni (Eds.), *Multi-Agent Programming: Languages, Platforms and Applications*. Springer.

Poshyvanyk, D. & Marcus, A. (2007). Using traceability links to assess and maintain the quality of software documentation. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07)*, Lexington, KY, USA.

Ramesh, B. & Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering, 27*(1), 58-93.

Rao, A. & Georgeff, M. (1992). An abstract architecture for rational agents. *Proceedings of Knowledge Representation and Reasoning (KRgR-92)*, 439-442.

Rao, A. S. & Georgeff, M. P. (1995). BDI agents: From theory to practice. *Proceedings of First International Conference on Multi-Agent Systems*, 312-319.

Rao, A. S. & Georgeff, M. P. (1998). Decision procedures for BDI logics. *J.Log.Comput., 8*(3), 293-342.

Ravichandar, R., Arthur, J., & Pérez-Quiñones M. (2007). Pre-requirement apecification traceability: bridging the complexity gap through capabilities. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07).*

Reiss, S. P. (2006). Incremental maintenance of software artifacts. *IEEE Transaction on Software Engineering*, 32(9), 682-697.

Rilling, J., Witte, R., & Yongang Zhang Y. (2007). Automatic traceability recovery: An ontological approach. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07).*

Rumbaugh, J., Booch, G., & Jacobson I. (1999). *Unified Modelling Language Reference Manual*. Addison-Wesley.

Sharif, B. & Maletic J. (2007). Using fine-grained differencing to evolve traceability links, *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07).*

Sherba, S. & Anderson, K. (2003). A framework for managing traceability relationships between requirements and architectures. *Proceedings of the Second International Workshop From Software Requirements to Architecture*.

Sherba, S. (2005). *Towards automating traceability: An incremental and scalable approach*. (Unpublished doctoral dissertation). University of Colorado at Boulder, Boulder, USA.

Spanoudakis, G., Zisman, A., Pérez-Miñana, E., & Krause, P. (2004). Rule-based generation of requirements traceability relations. *Journal of Systems and Software, 72*(2), 105-127.

Spanoudakis, G. & Zisman, A. (2005). Software traceability: A roadmap. In S. Chang (Eds.), *Advances in Software Engineering and Knowledge: Recent Advances, Vol III*. World Scientific Publishing.

Spanoudakis, G., Garcez A., & Zisman A. (2003). Revising rules to capture requirements traceability relations: A machine learning approach. *Proceedings of the 5th International Conference in Software Engineering and Knowledge Engineering*.

Standish Group. (2003). *What are your requirements?* Standish Group.

Sudeikat, J., Braubach, L., Pokahr, A., & Lamersdorf, W. (2004). Evaluation of Agent–Oriented software methodologies – examination of the gap between modeling and platform. *Proceedings of the Workshop on Agent Oriented Software Engineering*.

TAOME4E. (2008). *Tool for agent oriented modeling*. Retrieved March 2008, from http://sra.itc.it/tools/taom4e/.

ten Hove, D., Göknil, A., Kurtev, I., van den Berg, K., & de Goede, K. (2009). Change impact analysis for SysML requirements models based on semantics of trace relations. Proceedings of the ECMDA Traceability Workshop, ECMDA-TW 2009, 17-28.

TopicMaps.Org. (2001). *XML Topic Maps (XTM) 1.0 Specification, Pepper,* S. and Moore, G. (Eds), Retrieved from http://www.topicmaps.org/xtm/1.0/

Toranzo, M. & Castro, J. (1999). A comprehensive traceability model to support the design of interactive systems. *Proceedings of the II Workshop on Interactive System Design and Object Models.*

Toranzo, M., Castro, J., & Mello, E. (2002). Uma proposta para melhorar o rastreamento de requisitos. *Proceedings of the Workshop on Requirements Engineering.*

van den Berg, K., Conejero, J. M., & Hernández, J. (2006). Analysis of crosscutting across software development phases based on traceability. *Proceedings of the 2006 International Workshop on Early Aspects at ICSE*, Shanghai, China. 43-50.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. Proceedings of the Fifth IEEE International Symposium on Requirements Engineering.

Vanhooff, B. & Berbers, Y. (2005). Supporting modular transformation units with precise transformation traceability metadata. *Proceedings of the ECMDA Traceability Workshop.*

Wooldridge, M. (2002). *An introduction to multiagent systems*. John Wiley & Sons Ltd.

Wooldridge, M. & Jennings, N. (1995). Intelligent agents: theory and practice. *Knowledge Engineering Review*, 115-152.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. The MIT Press.

Wooldridge, M., Jennings, N., & Kinny D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems, 3*(3), 285-312.

WordNet. (2010). *About WordNet*. Retrieved December, 2010, from http://wordnet.princeton.edu/

XML. (2010). *XML Techonology*. Retrieved December, 2010, from http://www.w3.org/standards/xml/

XQuery. (2010). *Query*. Retrieved December, 2010, from http://www.w3.org/standards/xml/query

Yu E. (1995). *Modelling Strategic Relationships for Process Reengineering*. (Unpublished doctoral dissertation). Department of Computing, University of Toronto, Toronto, Canada.

Zisman, A., Spanoudakis, G., Pérez-miñana, E., & Krause, P. (2002) Towards a Traceability Approach for Product Families Requirements. *Proceedings of the 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications*.

Zisman, A., Spanoudakis, G., Pérez-miñana, E., & Krause, P. (2003). Tracing software requirements artefacts. *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03).*

Zou, X., Settimi, R., & Cleland-Huang, J. (2006). Phrasing in dynamic requirements trace retrieval. *Proceedings of the 30th Annual International Computer Software and Applications Conference*.

Zou X., Settimi R., & Cleland-Huang J. (2007). Term-based enhancement factors for improving automated requirement trace retrieval. *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07).*