RMIT UNIVERSITY

A SYSTEMATIC APPROACH FOR DETECTING FAULTS IN AGENT DESIGNS

A thesis submitted for the degree of

Doctor of Philosophy

Yoosef Bassam. Abushark

BSc. (Computer Science),

MSc. (Computer Science),

School of Science (Computer Science and Software Engineering),

College of Science, Engineering, and Health,

RMIT University,

Melbourne, Victoria, Australia.

Supervisors:

Assoc. Prof. John Thangarajah

Assoc. Prof. James Harland

Dr. Timothy Miller (The University of Melbourne)

March, 2017

Declaration

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

Yoosef Bassam. Abushark

School of Science (Computer Science and Software Engineering)
RMIT University
March, 2017

Acknowledgments

The PhD has been a journey of enormous learning for me, not only in the research arena, but also on a personal level. I would like to sincerely express my gratitude to the people who have supported me throughout the period of my candidature.

First and foremost, I would like to thank the supervisory team, Assoc. Prof. *John Thangarajah*, Assoc. Prof. *James Harland* and Dr. *Timothy Miller*, at the University of Melbourne. I would like to deliver my sincere appreciations to all of them for the encouragement and guidance they have provided me throughout my candidature. Their assistance and insightful advises helped me to overcome not only the challenges I faced in my studies, but also the ones I encountered on a personal level. I've been privileged to work with such a great supervisory team. Also, I'm very grateful to Prof. *Michael Winikoff*, at the University of Otago, for honouring me with the co-authorship and his valuable comments.

I also would like to thank the Agent Group at RMIT University. The academic activities that took place in the group have broadened my vision beyond my research arena. In particular, I'd like to thank Dr. *Rick Evertsz*, Dr. *Nitin Yadav* and Assoc. Prof. *Sebastian Sardina* for useful conversations and valuable suggestions.

Many thanks also should be given to my sponsoring institution, King Abdulaziz University, for granting me this extraordinary opportunity to continue my studies at RMIT University.

Most importantly of all, I owe a heartfelt thanks and gratitudes to my beloved wife, *Wardah Tayeb* and our son *Yazn*. I sincerely appreciate your sacrifices and your efforts in providing me with the proper aura for studying. *Wardah* has been supportive and considerate throughout my studies in Australia since 2011, without her support I could not persist in my research until the submission. I cannot forget to thank my sister *Essra Abushark* for believing in my ability and being always in touch with me throughout my residency in Australia. I also would like to thank my father for teaching me how to survive in my life and being independent. Also I'm grateful to my father-in-law *Adnan Tayeb* for being patient and considerate.

I also would like to acknowledge *Elite Editing* for proofreading my thesis. The editorial intervention was restricted to Standards D and E of the Australian Standards for Editing Practice.

Thanks to you all !

Dedication

I gratefully dedicate this thesis to the souls of my mother, Bothina Thabit (1952 - 2006), and my mother-in-law, Radeen Mohammed (1960 - 2014), may god have mercy on their souls.

Credits

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

- Y. Abushark, J. Thangarajah, T. Miller, J. Harland, and M. Winikoff. *Early detection of design faults relative to requirement specifications in agent-based models.* In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, *May 4-8, 2015*, pages 1071–1079. ACM, 2015. (CORE: A*)

- Y. Abushark, J. Thangarajah, T. Miller, and J. Harland. *Checking consistency of agent designs against interaction protocols for early-phase defect location.* In A. L. C. Bazzan, M. N. Huhns, A. Lomuscio, and P. Scerri, editors, International conference on Autonomous Agents and Multi-Agent Systems, AAMAS 14, Paris, France, *May 5-9, 2014*, pages 933-940. IFAAMAS/ACM, 2014. (CORE: A*)

- Y. Abushark, J. Thangarajah, T. Miller, M. Winikoff, and J. Harland. *Requirements specification in the prometheus methodology via activity diagrams.* In C. M. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, editors, Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems, Singapore, *May 9-13, 2016*, pages 1247–1248. ACM 2016. (CORE: A*)

- Y. Abushark. *A systematic approach for detecting defects in agent designs: (doctoral consortium).* In C. M. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, editors, Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems, Singapore, *May 9-13, 2016*, pages 1520–1521. ACM, 2016. (CORE: A*)

- Y. Abushark and J. Thangarajah. *AUML protocols: from specification to detailed design.* In M. L. Gini, O. Shehory, T. Ito, and C. M. Jonker, editors, International conference on Autonomous Agents and Multi-Agent Systems, AAMAS 13, Saint Paul, MN, USA, *May 6-10, 2013*, pages 1173–1174. IFAAMAS, 2013. (CORE: A*)

- Y. Abushark, M. Winikoff, T. Miller, J. Harland, and J. Thangarajah. *Checking the correctness of agent designs against model-based requirements*. In T. Schaub, G. Friedrich, and B. OSullivan, editors, ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), volume 263 of Frontiers in Artificial Intelligence and Applications, *pages 953–954*. IOS Press, 2014. (CORE: A)

Book Chapters:

- Y. Abushark and J. Thangarajah. *Propagating AUML protocols to detailed design.* In M. Cossentino, A. E. Fallah-Seghrouchni, and M. Winikoff, editors, Engineering Multi-Agent Systems - First International Workshop, EMAS 2013, St. Paul, MN, USA, *May 6-7*, 2013, Revised Selected Papers, volume 8245 of Lecture Notes in Computer Science, pages 19–37. Springer, 2013

Journals:

- Y. Abushark, T. Miller, J. Thangarajah, M. Winikoff, and J. Harland. *Requirements specification via activity diagrams for agent-based systems.* Autonomous Agents and Multi-Agent Systems, vol.31, issue 3, 423–468, 2017. (ERA: A)

- Y. Abushark, J. Thangarajah, T. Miller, and J. Harland. A framework for automatically ensuring the conformance of agent designs. Journal of Systems and Software, vol. 131, 266–310, 2017. (ERA: A)

The thesis was written in the TexStudio editor on Linux Ubuntu, and typeset using the $\LaTeX\,2_\varepsilon$ document preparation system.
All trademarks are the property of their respective owners.

Contents

List of Figures

List of Tables

Abstract

Multi-agent systems are increasingly being used in complex applications because of features such as autonomy, proactivity, flexibility, robustness and social ability. However, these very features also make verifying multi-agent systems a challenging task. Thus, techniques to detect and avoid defects in such systems are essential. In particular, it is desirable to detect defects as early as possible in the software development life-cycle (i.e., at the design phase). Nevertheless, little research has been conducted in this context.

This thesis proposes a mechanism, including automated tool support, for early-phase defect detection by comparing the plan structures of a belief-desire-intention (BDI) agent design against the following: (1) requirement models, specified in terms of scenarios and goals; and (2) agent communication models. The intuition of our approach is to extract sets of possible behaviour runs from the agent-behaviour models and to verify whether these runs conform to the specifications of the system-to-be. The proposed approach in this thesis is applicable at design time and does not require source code. Our approach is based on the Prometheus agent-design methodology but is applicable to other methodologies that support the same notions.

We evaluate the proposed verification framework on designs, ranging from student projects to case studies of industry-level projects. Our evaluation demonstrates that even a simple specification developed by relatively experienced developers is prone to defects, and our approach is successful in uncovering most of these defects. In addition, we conduct a scalability analysis of our methods, and the outcomes reveal that our approach can scale when designs grow in size.

CHAPTER 1

Introduction

Software systems based on autonomous agents are used in a significant number of applications in which the domain is highly dynamic [Munroe et al. 2006, Evertsz et al. 2014]. Agents in such systems are often autonomous so that they are able to make the most suitable decisions based on current environmental situations, without human intervention. Many models have been proposed for use as blueprints for developing agent-based systems [Wooldridge and Jennings 1994]. A popular and successful model that has been used in many application areas such as automated manufacturing, robotics and flight management is the *belief-desire-intention* (BDI) model [Rao and Georgeff 1995]. The BDI model has been used as the basis for many agent-implementation platforms such as JACK, Jason and JadeX [Bordini et al. 2006a].

The development of BDI-based systems involves many design and implementation activities; therefore, numerous *agent-oriented software engineering* (AOSE) methodologies have been proposed to act as frameworks to organise these activities (see [Winikoff and Padgham 2013] for a recent survey). The development activities within these methodologies are performed by software engineers (i.e., humans), which renders these activities error prone. These errors may result in undetected faults in the final product and may consequently lead to significant failures in the product's behaviour.

In software engineering, it has long been established that the *early* detection and resolution of software defects saves development costs, particularly for large projects [Boehm and Papaccio 1988, Boehm et al. 1981]. However, there has been little research into verifying agent designs against the specifications of the system-to-be (i.e., during the design phase and before the implementation phase). Existing work on verifying BDI-agent systems focuses on formal verification techniques [Dastani et al. 2009], particularly using model checking ([Dennis et al. 2012]) and theorem proving ([Sudeikat et al. 2006]) or run-time testing ([Padgham et al. 2013]). Although these techniques are effective for ensuring the correctness of the agent system, they perform a *late* check and require complete programmes to be written before any verification is performed.

In this thesis, we propose a framework grounded in the Prometheus agent-design methodology [Padgham and Winikoff 2005] that enables early (i.e., at the design phase) detection of defects in agent-based designs. Prometheus is a well-established methodology that has not only been adopted in academia, but is also used in the industry [Singh and Padgham 2015, Evertsz et al. 2015]. We use Prometheus, both due to our greater familiarity with the methodology and the accessibility to the source code of the methodology's tool-support (PDT). Being able to access the source code of PDT allowed us to integrate the tool-support of the proposed approach in this thesis with the methodology's tool. Although our framework is grounded in the Prometheus methodology, the general concepts and approach we use are applicable to other methodologies that support the BDI model of agency because they share similar notions [DeLoach et al. 2009].

The proposed *verification framework* checks that the agent-behaviour models conform to two *specification* models of the intended system: (1) high-level requirements in the form of scenarios and goal models, as specified in Prometheus; and (2) agent-communication protocols, which define the set of messages that can be sent between agents within the system. It is worth noting that our proposal exceeds the simple static consistency checks that tools such as the Prometheus Design Tool (PDT) [Padgham et al. 2005a] perform because our framework also considers the dynamic behaviour of the system.

We evaluate the proposed verification framework on designs, ranging from student projects to case studies of industry-level projects. These designs were developed by relatively experienced developers and based on the Prometheus methodology. The following industry-level projects are included in the evaluation: (1) an oil production simulator and (2) an air-traffic management system. Our results demonstrate that the approach is successful in revealing defects in agent designs with respect to the specification models of the sought system. For example, in the air-traffic management system, we revealed 15 defects (most of which were not detected via a manual check) in the agent designs concerning a large interaction protocol involving 11 participants.

Summary of Existing Work

In the context of AOSE, there has been little research conducted to verify the accuracy of BDI designs at the design phase. Existing work in verifying agent designs focuses on ensuring the static consistency between design entities across different design models ([DeLoach and Garcia-Ojeda 2010], [Cernuzzi and Zambonelli 2009] and [Padgham and Winikoff 2005]), or on validating the requirement models of the sought system ([Lopez-Lorca et al. 2016], [Giorgini et al. 2005] and [Fuxman et al. 2001]).

Lopez-Lorca et al. [Lopez-Lorca et al. 2016] propose an ontology-based approach to support the validation of the requirement models of an agent system. The approach is semi-automated and requires the following ontologies: (1) a domain ontology and (2) an agent-oriented meta-model. The approach then combines these two ontologies to validate, through automated reasoning, the requirement models of the agent-based system. Although this approach ensures that the requirement models of the sought system comply with the needs of the stakeholders of that system, it does not verify the agent-behaviour models against these requirement models.

Considering five of the most commonly used AOSE methodologies—Prometheus, Tropos, the Organisation-based Multi-agent Software Engineering (O-MaSE), INGENIAS and Gaia—only three provide support for agent-design verification and to a limited extent. Although these three AOSE methodologies partially support the verification of agent designs, they do not provide any support for verifying agent-behaviour models against requirement models.

Tropos supports the verification of the requirement models of the sought system. It offers two frameworks (each including tool support) for the following: (1) validating formal requirements specification ('T-Tool' [Fuxman et al. 2001]) and (2) reasoning with formal goal models ('GR-Tool' [Giorgini et al. 2005]). Although Tropos ensures the validity of the requirement models of the system-to-be, it does not support the verification of the agent-behaviour models (i.e., agents' low-level designs) against these requirement models.

The INGENIAS Development Kit (IDK) [Gómez-Sanz et al. 2008], which supports the INGENIAS methodology, integrates a tool named 'ACLAnalyser' for analysing the interaction specifications between agents by executing the system and logging the run-time interactions [Botía et al. 2006]. The tool uses the JADE code-generation module and, hence, the agent templates will not be generated if the models are lacking certain information. Although the IDK, through the ACLAnalyser tool, provides feedback about the interactions specified in the design phase, it does not provide any support for verifying agent-behaviour models against requirement models.

Prometheus, O-MaSE and Gaia methodologies provide agent engineers with a graphical environment for developing agent-based systems (PDT, [Padgham et al. 2008]; AgentTool III, [DeLoach and Garcia-Ojeda 2010]; Gaia4E, [Cernuzzi and Zambonelli 2009]). These tools do not offer any framework for verifying and validating the agent-behaviour models against the specifications of the *system-to-be*, although O-MaSE and earlier versions of PDT supported limited static consistency checking of the design. For example, PDT generates warnings such as events that are not handled and messages that should be sent/received according to a protocol. While these checks are based solely on static relationships between the design entities, our work assesses the dynamic behaviour of designs.

Research Questions

In this research, we develop a framework with tool support to verify agent-based design artefacts before the implementation phase. The novelty of this research is the ability to verify a variety of agent-based systems at the design phase before implementation. To achieve this goal, the following questions must be answered:

1. What *design artefacts* should be included in the verification process?

   The answer to this question identifies the specification models of the system under investigation and the design units to be considered in the proposed approach.

2. What is a suitable *executable model* for an agent-based system's specification models?

   In Prometheus, system-specification models are semi-formal. Thus, there is a need to transform these specification models to executable models with precise semantics so they can be used in the verification process. This question investigates different formalisms with the aim of using them as executable representations for the system specification models. Given that these models encompass both the requirements and the agent communications, this question has two sub-questions:

   (a) What is a suitable executable model for representing *requirement models*?
   (b) What is a suitable executable model for representing *agent communications*?

3. What is a suitable *executable model* for representing *agent-behaviour models*?

   Like the specification models in the previous question, agent-behaviour models in Prometheus are semi-formal; therefore, there is a need to transform these agent-behaviour models to executable models so they can be used in the verification process. In Prometheus, a realisation of a given specification model (scenario or protocol) is often spread across multiple agents. The steps of a scenario are associated with roles, which can be assigned to multiple agents. Similarly, agent communications often involve multiple agents and, hence, the realisations of such communications are often spread across the agents that are involved in these communications. In this question, we investigate how to merge all the relevant design entities of a given specification model into one coherent structure. We then investigate how the coherent structure can be translated into an executable model.

4. How can an *automated checking framework* be developed to *detect design defects effectively*?

   We address this question through the following two sub-questions:

   (a) How can all possible behavioural runs be extracted out of the agent-behaviour models?
   (b) How can the behavioural runs be checked against the specification models?

5. How can the proposed verification framework be evaluated?

   The aim of our evaluation is to ensure the effectiveness of the framework proposed by answering the following sub-questions:

   (a) Which design defects can the framework detect?
   (b) What is the level of false positives generated by the proposed approach?
   (c) How scalable is the proposed framework?

To the best of our knowledge, there are no existing benchmarks that can be used for evaluating the proposed approach. Thus, we conduct an empirical evaluation by applying this proposed approach to complete agent systems. As part of our evaluation, we have considered 15 designs. Some of these designs were developed as final projects in the agent-oriented programming and design course, at RMIT University, whereas others were developed by the agent group at the university[1]. In addition, we have included two industry-level case studies in the evaluation: (1) an oil production simulator and (2) an air-traffic management system. We have also conducted a scalability analysis on 24 synthesised plan graphs to measure the scalability of our approach.

We have used quantitative and qualitative measures in our evaluation, quantifying the number of errors detected and categorising the errors (true and false positives) for qualitative measures.

Summary of Contributions

The research contributions of this thesis are summarised below.

1. *Verification Framework*: The major research contribution for this thesis is the creation of a verification framework designed to ensure the correctness of the agent-behaviour models (the events and their associated plans) with respect to the specification (i.e., requirement models or agent-communication protocols) of the sought system. The framework is applicable at the design phase and before any implementation is performed. We believe our proposal will have a positive effect on the development costs of agent-based systems. More importantly, our approach contributes to the widespread use of agent technology because it increases the reliability of such systems.

2. *Automated approach for generating activity diagrams from scenarios and the goal model*: A further contribution of this research project is the addition of activity diagrams to complement the process of specifying requirements in Prometheus. Our evaluation demonstrated that including activity diagrams leads to a better understanding of the requirement models of the sought system. This is achieved by providing a more holistic view of what the system is designed to achieve and how it should behave. Activity diagrams also provide assistance when performing maintenance on the sought system. Participants in our experiment unanimously agreed that the inclusion of the activity diagram improved their abilities to understand the re-

[1]The agent group is part of the intelligent systems area within the School of Science (Computer Science) at RMIT university.

quirement models. Therefore, we recommend that the Prometheus methodology—and, indeed, the other AOSE methodologies as well—include activity diagrams as an integrated feature.

3. *Tool-Support*: Through this research, we developed an Eclipse plug-in that integrates with PDT to automate our verification methods. Although the tool is grounded in PDT, we believe that some of the code packages can be reused as part of the supported tools of other AOSE methodologies.

Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background for understanding the various elements of this thesis, including the basics of the agent systems and the AOSE discipline. Further, this chapter presents the state-of-the-art of AOSE discipline and assuring the correctness of agent designs. Chapter 3 describes the conceptual model of the proposed verification framework, explaining the design units considered. This chapter also describes and documents the concepts of and assumptions about our verification framework. The chapter concludes by providing an overview of the technicalities of the verification framework. Chapter 4 explains the transformation of requirements in Prometheus into unified-modelling language (UML) activity diagrams. The current representation of requirements in Prometheus has some limitations. To overcome these limitations, we propose the use of the activity diagrams to complement the process of specifying requirements in Prometheus. The chapter also includes the user study that was performed as part of this research to evaluate our proposal. Chapter 5 presents the technical approach we used to transform the semi-formal models in Prometheus to executable models. This transformation includes the specification and agent-behaviour models in Prometheus. Chapter 6 explains the process of checking the correctness of the agent-behaviour models against a given specification model in the sought system (i.e., requirements models or agent-communication protocols). This chapter also discusses the tool support of the verification framework. Chapter 7 presents the evaluation of the proposed verification framework and a discussion of the approach created by this research. Chapter 8 concludes the thesis and details possible extensions of this work.

CHAPTER 2

Background and Related Work

In this thesis, we propose a suite of techniques, which can be used to detect defects in BDI systems during the design phase—that is, *prior to implementation*. This chapter explains the necessary background for understanding the various elements of this thesis. It also reviews relevant literature in the area of AOSE. Section 2.1 introduces some of the concepts and characteristics of agents and Section 2.2 explains a popular agent architecture—the *belief-desire-intention* architecture—that serves as a basis for agents in this thesis. In Section 2.3 we discuss the AOSE discipline, which is intended to help software engineers to plan such systems, along with the process of specifying requirements and agent interactions. Section 2.4 details the Prometheus methodology, which forms the groundwork of our approach. Section 2.5 briefly introduces some other, commonly used methodologies to show that they share similar notions for specifying and designing agent-based systems.

In the proposed approach, Petri nets are used as a formal representation of the system's specification and to extract all possible behaviour runs from the agent-behaviour models. Additionally, we propose using UML-activity diagrams as more structured representation of requirement models to complement the process of specifying requirements in the Prometheus methodology. Section 2.6 gives an introduction to Petri nets and Section 2.7 introduces the UML-activity diagrams.

Section 2.8 introduces the concepts of software verification and validation as a step in developing agent-based systems. The final section (Section 2.9) reviews existing work in verifying software systems, including agent-based systems, both automatically and manually.

2.1 Agent Systems

In the context of this thesis, an *agent system* is *goal-oriented computer system* that acts on behalf of the user to achieve the goals desired. The higher-level of modelling abstraction offered by this paradigm helps system designers to manage complexity and, hence, to improve the construction of complex systems [Jennings 2001]. These systems exhibit a set of characteristics that make this paradigm suitable for building systems that operate in highly dynamic and unpredictable environments. The characteristics are as follows [Wooldridge and Jennings 1994]:

- *Autonomy:* Entities in the system (agents) behave independently and without any intervention. They are responsible for taking their decisions based on their current state and their environment.
- *Social ability:* In multi-agent systems, agents have the ability to interact to pursue their desired goals. Such socialisation is modelled as communications between different agents within the system. Thus, inter-agent communication plays a significant role in achieving the systems' goals. For example, in an agent-based auction system, the buyer agent needs to communicate with the seller agent to complete the sale process.
- *Reactiveness:* Agents continuously perceive their environment and respond to any changes in a timely manner.
- *Pro-activeness:* Agents are persistent in pursuing their goals irrespective of changes in the environment. Note that a combination of *pro-activeness* and *reactiveness* is often a source of conflict.

2.2 The BDI Architecture

An agent architecture is an abstract model of an agent-based system. Many architectures have been proposed as blueprints for developing agent-based systems [Wooldridge and Jennings 1994]. This thesis focuses on a popular architecture, the *belief-desire-intention* (BDI) architecture. The roots of this architecture are found in a philosophical model proposed by Bratman [Bratman 1987, Bratman et al. 1988, Bratman 1999] and formalised by Rao and Georgeff [Rao and Georgeff 1991;

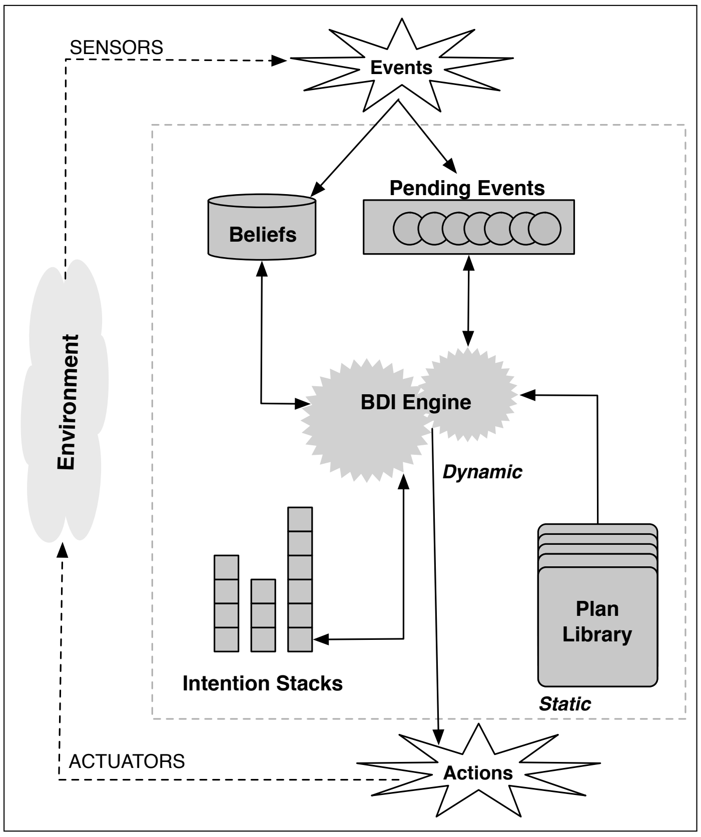


Figure 2.1: Execution cycle in the belief-desire-intention architecture

1995]. This model supports the notion of agency by having each agent within the system uses its beliefs, desires and intentions [Kinny et al. 1996].

Beliefs are what an agent believes about itself and its surrounding environment. The term *'beliefs'* is used rather than *'knowledge'* or *'fact'*, since the information an agent believes is not necessarily always true. For example, consider a *soccer player* agent 'A', who wants to pass the ball to player agent 'B'. Agent 'A' in this case believes that it has the ball (beliefs about itself) and believes that agent 'B' is beside it (beliefs about the environment). However, it is not necessarily true that agent 'A' has the ball when it intends to pass the ball, or that agent 'B' will be beside it.

Desires are what an agent wants to achieve within the system [Rao and Georgeff 1992]. To be achievable, the desires should be consistent and feasible. For example, a *soccer player* agent cannot pursue its goals to pass a ball while believing that it does not have the ball.

Intentions are ways of achieving agents' goals. For example, a *soccer player* agent, who believes it has the ball, intends to pursue a goal to pass the ball to the nearest player. The agent needs to execute the proper steps that enable it to attain this goal. First, it may intend to explore the field for locating the players next to it, and then it decides the nearest player according to its beliefs. It will then adopt the intention to pass the ball.

A BDI-based system is *situated* in a dynamic environment and continually interacts with it by handling all inputs (*events*) as well as producing outputs (*actions*) that may change the state of the environment. The inputs can be received from the environment (external), or they can be internal events within the system. The system has a library that contains a collection of *plans* to serve as recipes for handling possible inputs to the system.

Figure 2.1 shows the overall architecture of a BDI-based system and its execution cycle. As the figure depicts, the system perceives the inputs from the environment through sensors. These inputs may change the beliefs of the agents in the system and, consequently, affect the selection of the plan to handle the input from the intention stack. If a plan is selected, the agent will commit to execute it to satisfy its trigger (i.e., the event that triggered the plan). The final result of the execution of a plan could involve a series of actions affecting the environment and may produce other inputs (events) to the system.

Many BDI-based implementation platforms have been proposed, including JACK [Winikoff 2005b], Jadex [Pokahr et al. 2005] and Jason [Bordini et al. 2007]. These platforms support the BDI execution cycle explained by Rao and Georgeff in [Rao and Georgeff 1995].

Events and Plans

In most BDI-based platforms, intentions are realised by *plans*, while *goals* are captured by *events* or *triggers* [Winikoff 2005b]. Events represent inputs to the system and they can be external (perceptions of the environment) or internal. These *events* are initiators that stimulate agents to respond to them accordingly by executing the appropriate plans. Plans act as *recipes* that state the steps for addressing their associated events [Winikoff et al. 2001]. Even though the implementation of plans is different from one platform to another, they share a common structure: *name, body, invocation condition (trigger)* and *precondition*. The name of a plan is a unique string that identifies the plan in the system. The body of a plan is the course of steps that enables the agent to satisfy the event that triggered it. A plan may invoke other plans to realise other goals and

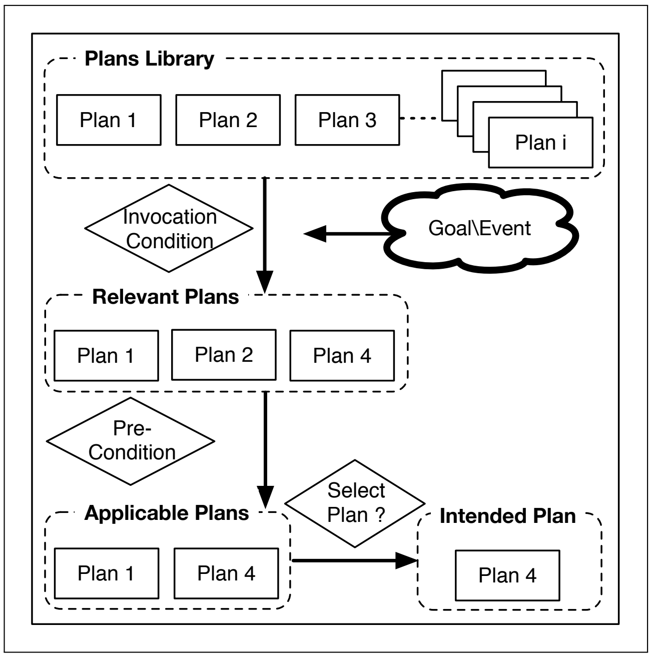


Figure 2.2: Determination mechanism for plans in the bdi model

to achieve a successful execution.

The invocation condition of a plan is a condition that determines whether the plan is relevant to a given event or not. However, each plan has a different precondition for determining the applicability of the plan to its triggering event, considering the current state of the agent ([Wooldridge 2000], page 29).

Considering the *soccer player* agent, let us assume that the event is ‘Handle_Ball’, and that there are two plans: (1) ‘Pass_Ball’ and (2) ‘Shoot_Ball’. Additionally, let us assume that the ‘Handle_Ball’ event is the trigger for both plans, whereas the precondition for each plan is different. Regarding the ‘Pass_Ball’ plan, the precondition is that the distance to the *goal keeper* should be greater than 10 metres. The precondition of the ‘Shoot_Ball’ plan is that the distance to the *goal keeper* should be less than 10 metres. Given this, once the ‘Handle_Ball’ event triggers both plans, in the system, one plan will be executed (i.e., deemed applicable) based on the distance of the player agent from the goal keeper.

Figure 2.2 demonstrates the BDI-agent execution cycle. Assuming that a multi-agent system is developed according to the BDI model, at some point an event will occur in the system (either from a percept or internally) and the system will handle it. In this case, the agent responsible for handling that event would react. As shown in Figure 2.2, the agent's plan library will be filtered to reveal relevant plans for the particular event. Based on the plan's context (the *precondition*), the relevant plans list will be filtered to reveal the applicable plans list. Then, the agent will select a plan from the applicable plans list to execute.

2.3 Agent-oriented Software Engineering

AOSE is a discipline that helps software engineers in conceptualising, designing and implementing agent-based software systems. The agent-oriented paradigm is of interest not only as a research area but also in industry, since it is suitable for building complex systems ranging from critical systems used in crisis management [Murakami et al. 2002] to systems such as information exchange [Munroe et al. 2006]. It is also suitable for systems that need to operate in a dynamic and unpredictable environments [Wooldridge 1998, Pechoucek and Marík 2008].

The development of agent-based systems involves many design and implementation activities. Numerous AOSE methodologies have been proposed to provide the necessary frameworks to organise these activities. MaSE [Deloach 2004], ROADMAP [Juan et al. 2002], Tropos [Bresciani et al. 2004], INGENIAS [Pavón et al. 2005], Gaia [Wooldridge et al. 2000, Zambonelli et al. 2003], PASSI [Cossentino 2005] and Prometheus [Padgham and Winikoff 2005] are some of the most commonly used AOSE methodologies for developing BDI-agent systems. In addition, there are many frameworks that allow process engineers to create custom agent-oriented methodologies, such as FAML [Beydoun et al. 2009] and O-MaSE [DeLoach and Garcia-Ojeda 2010].

Each AOSE methodology consists of a number of phases in which activities take place that result in many design *artefacts*. Design artefacts document information about the development process including requirement specifications and agent-behaviour models and, thus, they are considered the *blueprints* for the proposed system. While these methodologies differ in many ways, they are common in their inclusion of the core activities, namely: analysis, design, implementation and testing. The first two activities (analysis and design) are the focus of this thesis. In the analysis activity, the specifications (i.e., the requirements) of the intended system are to be determined. These requirements are designed in a concrete and well-structured way to provide a foundation for the implementation activity. The following subsections investigate the specification of these requirements in the context of AOSE paradigms and the relevant design artefacts.

2.3.1 Requirement Engineering in AOSE

A fundamental aspect of any software engineering methodology is the specification of requirements and the related area of requirement engineering (RE). Requirement engineering is a process that provides systematic techniques for ensuring the quality of the system requirements ([Sommerville and Sawyer 1997], page 5). This process encompasses a number of activities including requirements elicitation, analysis, specification, verification and management (i.e., managing any change in the requirements). Many RE methodologies have been proposed. While these RE methodologies are common in their inclusion of the basic activities, they may vary in terms of techniques applied [Misra et al. 2005]. In the context of the agent-oriented paradigm, most of these approaches are centred on the concept of *goals* for capturing agent-based requirements [van Lamsweerde 2000].

In goal-oriented RE, the intended system is defined and analysed in terms of goals, which are the objectives that the system must attain ([van Lamsweerde 2009], page 259). These goals are classified into two categories according to type of concern [van Lamsweerde 2001]: first, the functional requirements that concern the services provided by the system; second, the non-functional requirements that specify the quality of these services, such as the reliability of the system. These goals can be modelled through the following: (i) their types (e.g., functional and non-functional), (ii) their attributes, such as the priorities of the goals; and (iii) their associations with other entities in the model including the inter-goals links. The main purpose of these modelling techniques is to facilitate reasoning about goals. In the context of the agent-oriented paradigm, most of the methodologies use goals when specifying requirements [DeLoach et al. 2009].

2.3.2 Agent Communication

In multi-agent systems, communication plays a significant role, since it enables agents to exchange information. Agents communicate through *message passing* and, hence, it is important to ensure that they can understand each other. In *open systems*, there is a need to unify the language to make sure that agents from different systems understand each other. Thus, *agent-communication* *languages* (ACLs) were proposed to standardise communication between agents by providing formats and semantics for the messages exchanged. There are two main ACLs in the literature: *knowledge query and manipulation language* (KQML) [Finin et al. 1994] and the *Foundation for Intelligent Physical Agents* (FIPA) [Fipa 2002]. Messages in both languages consist of two parts ([Wooldridge 2009], page 106): (1) an *outer* part and (2) a *content* part. The outer part of a message (known as an 'envelope') defines the format of the message being sent, stating its *preformative* status (the class of the message) and its *parameters*. The content of the message being sent needs to be represented in a specific language, such as the *knowledge interchange format*.

The communication between agents is modelled using *interaction protocols*. An interaction protocol strictly defines the ways in which agents ought to communicate to accomplish their objectives in a joint setting. Protocols facilitate *negotiation* and argumentation between agents in multi-agent systems [Aknine et al. 2004, Artikis et al. 2007]. They are useful in many scenarios, such as the *auctioning* scenario, where the auctioneer agent needs to communicate with the bidder agents to complete a sale process [David et al. 2002].

Many approaches are used to represent protocols including the finite state machine approach used in the work of electronic institutions [Arcos et al. 2005], statecharts [Paurobally et al. 2003], Petri nets [Cost et al. 2000] and standard UML [Odell et al. 2001b], as well as its extended version to comply with agent standards (AUML) [Odell et al. 2001a]. The remainder of this section concentrates on the AUML notations, not only because popular AOSE methodologies have adopted them, but also because they are employed by the Prometheus methodology, the AOSE methodology used in this thesis.

Modelling interaction protocols with AUML

AUML is an extension of the standard UML used in the object-oriented paradigm [Odell et al. 2001b]. The process of creating interaction protocols involves generalising interaction diagrams that show the flow of messages between agents in the system to encompass interaction protocols using the AUML notations. Some of the most commonly used *interaction diagrams* in the modelling of interaction protocols are *AUML-sequence diagrams* [Huget and Odell 2004]. AUML-sequence diagrams, which are used to depict the interactions between agents visually, are the same as the UML sequence diagrams that are used in the object-oriented paradigm. However, instead of having objects as the main entities of the diagram, agents—or agents roles—form the main entities [Padgham et al. 2005a].

AUML-sequence diagram

An AUML-sequence diagram, also called a *protocol diagram* [Bergenti and Poggi 2000], a dynamic AUML model [Odell et al. 2000] showing the flow of messages between agents in an interaction protocol. Sequence diagrams not only demonstrate the communication between agents but also restrict the order of their messages through a number of constructs [Huget and Odell 2004]. Constructs are entities that control the sequence diagram's execution flow. The AUML-sequence diagram has eight different constructs, as follows [Winikoff 2005a]:

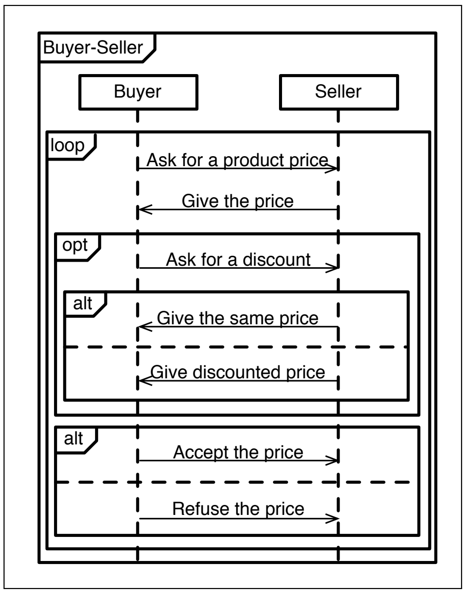
- Alternative construct [alt]: indicates that one of the multiple regions is executed based on the region's guard (the condition that must be true for the region to be executed).
- Option construct [opt]: indicates that communication may or may not occur based on the construct's guard.
- Loop construct [loop]: shows the repetition of a sequence of design entities (e.g., messages) for a fixed number of iterations by indicating a number or drawing on a Boolean expression.
- Break Construct [break]: shows that the communication has been interrupted.
- Stop construct [stop]: indicates the end of the agent's communication lifeline.
- Parallel construct [par]: allows the communication to be made in parallel.
- Reference [ref]: enables the designer to indicate another protocol within the modelled protocol by referring to the name of that protocol. Thus, when the execution flow comes to that reference construct, the referenced protocol will be executed.
- Continues [goto/label]: is used to control the execution flow of a protocol through two directives: goto and label. In other words, the designer can change the sequence flow to a specific point within the protocol by stating the goto directive along with the label's name, indicating the point within the protocol.

Directive	Description	Command Structure	Example
start	This directive indicates the start of the protocol and is used to declare the protocol's name.	start'Protocol's Name'	start A-B
agent	This directive is used to define an agent who is involved in the protocol.	agent 'Alias Name' 'Actual Name'	agent A Buyer
actor	This directive is used to define an actor who is external to the system and involved in the protocol.	actor 'Alias Name' 'Actual Name'	actor A Web-Server
box	This directive is used to declare the body of a construct (alt, loop, opt and par for alternative, loop , option and parallel constructs).	box 'Construct's Name'	box alt
next	This directive indicates the end of a region and the beginning of another region in a construct.	next	next
end	This directive indicates the end of a construct's body.	end 'Construct's Name'	end alt
message	This directive is used to define a message to be sent to another agent.	message 'Sender Agent's Alias Name' 'Receiver Agent's Alias Name' ' The Message'	message A B M1
action	This directive is used to define an action to be posted to the environment that is represented by an actor.	action 'Sender Agent's Alias Name' 'Receiver Actor's Alias Name' ' The Action'	action A B A1
percept	This directive is used to define a percept that is received from the environment to be handled by an agent.	percept 'Sender Actor's Alias Name' 'Receiver Agent Alias Name' 'The Percept'	percept A B P1
guard	This directive is used to define a condition for a message. Consequently, the message will not be sent until that condition is satisfied. The guard must be defined before the intended message.	guard ['The Condition']	guard [x = 5]
finish	This directive indicates the end of the protocol. The textual protocol must end with this directive.	finish	finish

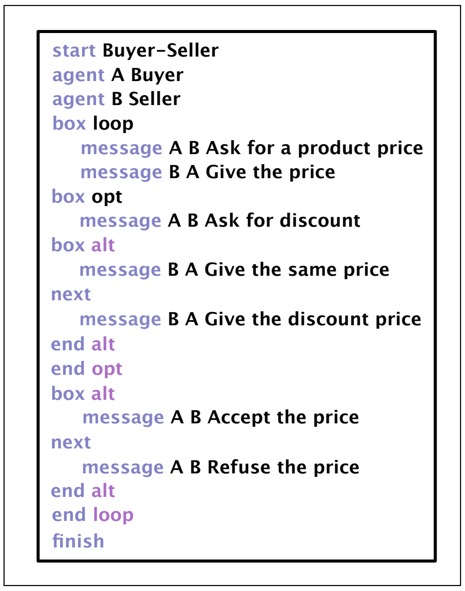
Table 2.1: AUML textual notation directives

AUML-sequence diagrams can be constructed in two ways, using either graphical or textual notations [Winikoff 2005a]. AUML textual notation is considered a convenient way to construct and model interaction protocols. Because of the simplicity of the textual AUML syntax, AUML textual format is fast and easy to write and edit [Winikoff 2005a]. To textually construct an AUML interaction protocol, the written AUML textual notations must be well structured according to the following rules:

- Having each AUML textual command on a separate line is essential.
- All the commands must be well nested. In other words, where an alternative construct is embedded in a loop construct, the body of the alternative construct must be ended before closing the loop definition.



(a) AUML-sequence diagram



```
start Buyer-Seller
agent A Buyer
agent B Seller
box loop
    message A B Ask for a product price
    message B A Give the price
box opt
    message A B Ask for discount
box alt
    message B A Give the same price
next
    message B A Give the discount price
end alt
end opt
box alt
    message A B Accept the price
next
    message A B Refuse the price
end alt
end loop
finish
```

(b) AUML textual notation

Figure 2.3: Buyer-Seller interaction protocol

- All the AUML textual notation directives must be written in lowercase format (see Table 2.1).
- The protocol must begin with the keyword start to declare the protocol's name.
- After declaring the protocol's name, all the involved participants (i.e., agents and actors) must be declared by using the agent/actor directive.
- The protocol must end with the finish keyword to indicate the end of the protocol.

AUML-sequence diagram example

This example assumes that a virtual shop is developed as a multi-agent system. The system has two agents: a buyer and a seller. The agents need to communicate to achieve a successful sale. Figure 2.3 shows the AUML-interaction protocol between the buyer agent and the seller agent along with its textual notations. The following scenario illustrates the communication between the buyer agent and the seller agent.

The buyer asks the seller for a product's price, then the seller replies by giving the buyer the price. Optionally, the buyer agent may ask for a discount. In the case where the buyer asks

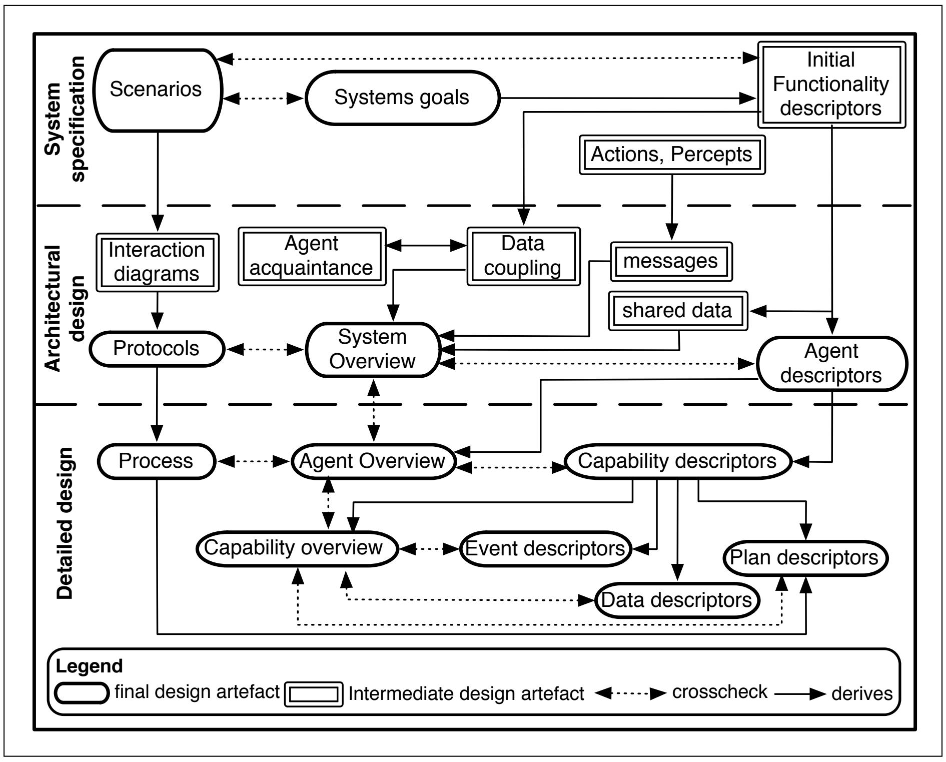


Figure 2.4: Phases of the Prometheus methodology

the seller for a discount, the seller agent may respond by offering either the same or a discounted price. The buyer may then accept the price or refuse it. This process may be repeated.

2.4 The Prometheus Methodology

Prometheus is a well-established methodology that provides support for the complete agent-based software development life-cycle [Padgham and Winikoff 2005]. It provides software designers with a process associated with artefacts, resulting in a well-documented design. The methodology consists of three phases: the *system specification* phase, the *architectural design* phase and the *detailed design* phase. The output of each phase in Prometheus is delivered as an input into the next phase (see Figure 2.4). The following briefly introduces the relevant parts of Prometheus to the work in this thesis, using a transport ticketing smart-kiosk as a running example.

Transport-ticketing Smart Kiosk

Transport-ticketing smart kiosk is an agent-based system that implements many ticketing transactions including *top up, refund, journey planning, account inquiries, lost cards* and *card replacement*. The system has four agents as follows:

- **Ticketing agent**: this agent is responsible for managing the ticketing transactions. This agent is at the front end—in other words, interacting with customers by finalising their requests. This agent is also responsible for updating the provider servers as to the transactions that take place in the system.
- **Payment manager**: this agent is responsible for managing sale transactions such as payments and refunds.
- **Scheduling agent**: this agent is responsible for tracking live transportation data, such as departure times, arrival times and changes and delays in the transportation system.
- **GUI (graphical user interface)**: this agent is responsible for displaying messages from the other three agents on the touch screen and all possible transactions for customers.

The system interacts with their clients through seven interfaces: the radio-frequency identification (RFID) reader and writer, the transportation provider server, the touch screen, the card slot, the printer, the money scanner and the banking server.

The customer should have a smart pass card (an RFID card). This card is then used as an input into the system (through the RFID reader/writer port). Using this smart pass card, a customer can interact with the system using a touch screen. The customer should be able to conduct the following transactions: *top up, lost-card, plan journey, make account inquiries, request refund, purchase a new card* and *replace card*. For the purpose of explaining the relevant parts of the methodology, we will consider only one transaction out of the seven mentioned above. Specifically, we will describe the *lost card* scenario.

- **Lost card**: the system should enable customers to process their lost card requests by either issuing a new card or refunding the amount on the lost card. Customers should be allowed to provide their subscription number and password using the touch screen. After validating the customer's credentials, the system should allow the customer to pick one option out of the following two: (1) issuing a new card or (2) refunding the amount. The refund can be conducted as follows: (1) direct deposit into the linked bank account or (2) in cash. If the customer chooses to have a new card issued, the system should be able to: (1) transfer the account information onto the new card, (2) update the provider servers with the details of the new card and (3) write the info to the card before releasing it to the customer (through the card slot).

If the customer chooses to have the amount refunded, the system needs to execute an interaction protocol with five participants: the ticketing agent, the payment manager, the GUI, the banking server (actor) and the money scanner (actor). Initially, the payment manager should receive a refund request from the ticketing agent before obtaining the current balance. The payment manager should act according to the refund option chosen. If the deposit option has been selected, the payment manager should (1) obtain the details of the customer's bank account that is linked to the card; (2) deposit the amount on the lost card into the bank account; (3) receive notification through the system once confirmation has been obtained from the bank server. If the cash option has been selected, the payment manager should check the cash availability first. In the case that sufficient cash is available, the money scanner will be commanded to release the amount and then to confirm the delivery. If this does not occur, the ticketing agent should be informed. At the end of this process, a notification will be sent to both agents: the GUI and the ticketing agent. The ticketing agent will print a receipt and the system will update the provider servers with the changes. If the amount on the lost card is less than or equal to zero, the system will notify the customer and abort the transaction.

System specification phase

In the system specification phase, the requirements are taken as an input and the initial specification of the system is drawn, defining the goals to be pursued and the scenarios, which are the basic runs that the system-to-be should capture. Additionally, the external entities (actors), system inputs (percepts), and system outputs (actions) of the intended system are defined. The primary outputs from this phase are a goal-overview diagram, a definition of the interface between the system-to-be and its environment and a collection of scenarios (see Figure 2.4).

Revisiting the running example (lost card scenario), the system should interact with the environment through an interface that encompasses three actors: the *RFID reader/writer, the touch*

Type		Name	Role
1	Percept	LostRequest	Issuer
2	Goal	ExtractRequestInfo	InfoExtractor
3	Goal	VerifyCredentials	Issuer
4	Action	EnteredCredentials	Issuer
5	Percept	AccountInfo	Issuer
6	Goal	GetAccountInfo	Issuer
7	Goal	RefundOldCard	Seller
8	Goal	UpdateServersWithLostRequest	Updater
9	Action	UpdateAccountData	Updater

Figure 2.5: Lost card (refund option) scenario description

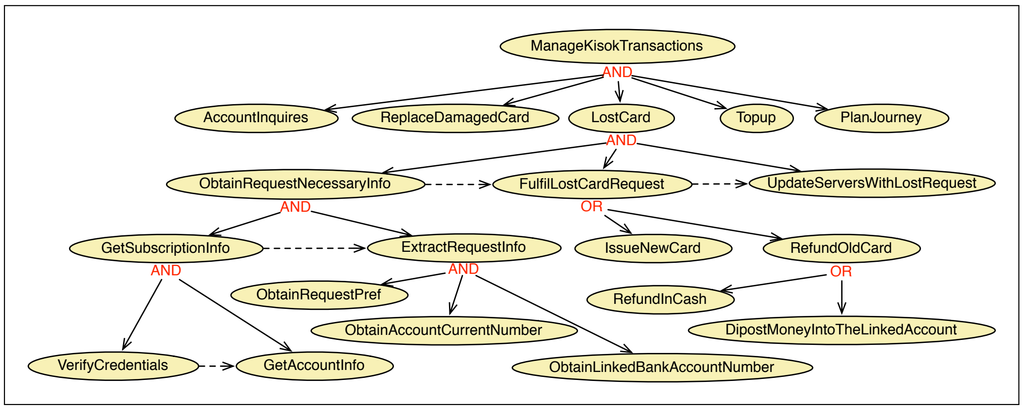


Figure 2.6: Goal overview diagram for the transport-ticketing smart kiosk

screen and the *banking server*; percepts: *LostRequest* and *AccountInfo*; and should interact by posting the following actions: *UpdateAccountInfo* and *EnteredCredentials*.

In the Prometheus methodology, scenarios are a sequence of steps. A step in a scenario can be one of the following types: an action (i.e., something the agent does), a percept (i.e., an input from the environment), a goal to realise or a sub-scenario. Each step is associated with a number of roles. Revisiting our running example, the *lost card request* transaction can be fulfilled either through refunding the card or issuing the customer with a new card. Figure 2.5 shows a particular run (i.e., scenario) that the system-to-be should implement; namely, the *lost card request* scenario. This scenario enacts the refund option without the issue of a new card. Note that the aim of the scenario is to capture a sample trace through the system's behaviour and it therefore does not specify a complete set of execution traces. However, as we shall see in Chapter 4, the information in scenarios and goal-overview diagrams can be used to construct constraints that must be met by

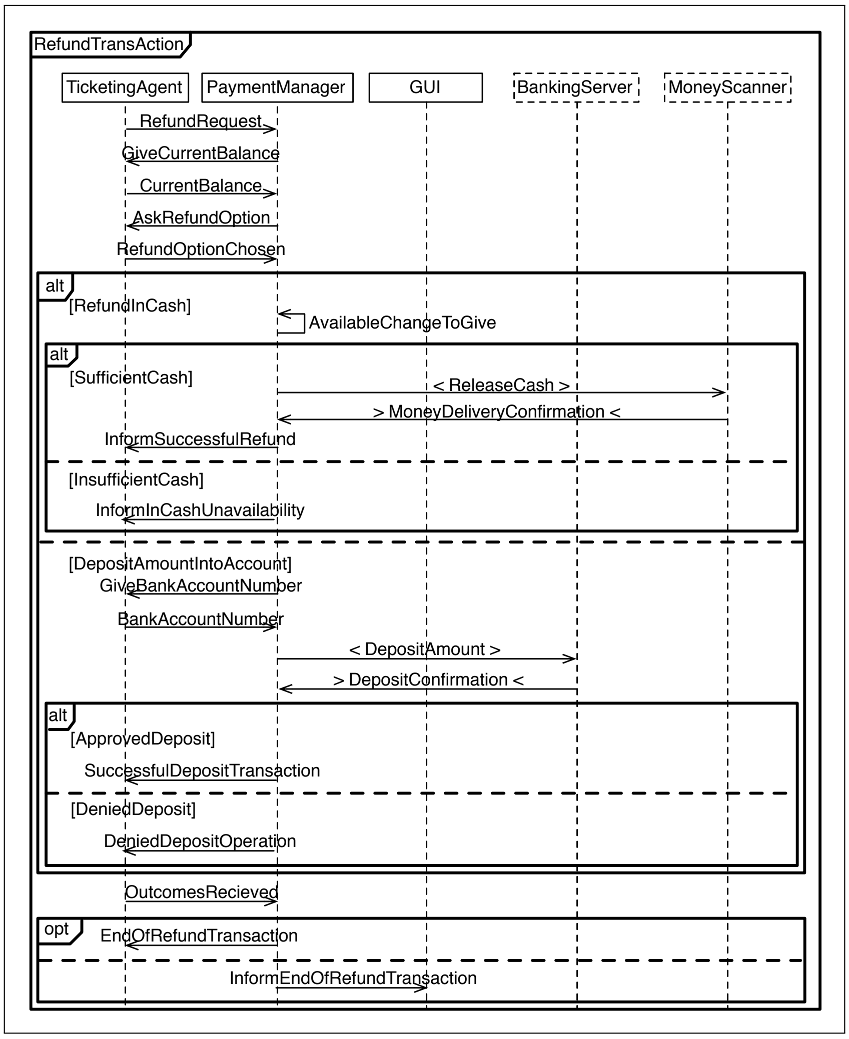


Figure 2.7: Refund interaction protocol

the agents' designs.

Goals are commonly modelled using a *goal diagram* that shows the relationship between goals, including how goals are decomposed into sub-goals. There are three types of goal decompositions (shown in Figure 2.6): disjunctive, undirected conjunctive or directed conjunctive. The disjunctive decomposition (denoted by OR) implies that a parent goal is realised if any of its children are realised. The undirected conjunctive decomposition (denoted by AND) means that a parent goal is realised if all its children are realised in some unspecified order. The directed conjunctive decomposition (denoted by AND with dashed arrows between the children) implies that a parent goal is realised if all its children are realised in the specified order. The dashed arrows between the children indicate the ordering constraints (e.g., *VerifyCredentials* before *GetAccountInfo*). To distinguish between the two AND decompositions, we refer to the directed conjunctive as SEQ in this thesis. Figure 2.6 shows a goal-overview diagram[1] for the transport-ticketing smart kiosk that outlines the goals and sub-goals required to achieve a lost card request successfully.

Architectural design phase

The architectural design phase is concerned with the internal architecture of the system. Based on the system goals and scenarios from the previous phase (the system specification phase), the roles and agent types are determined; then, the interactions between agents are modelled. The overall system architecture—the agent types and their interactions—is captured in the system overview diagram forming the primary output of this phase. The system-overview diagram captures each agent type in the system linked to its data sets, the events they receive (percepts), the events they post (their actions) and communication protocols between agents. AUML textual notations are used to model the necessary interaction between agents involved in the scenario. The AUML textual notations are translated and presented in the AUML-sequence diagram.

According to the description of the lost card scenario, there should be an interaction between five participants: the ticketing agent (agent), the payment manager (agent), the GUI (agent), the banking server (actor) and the money scanner (actor). Figure 2.7 depicts the AUML-sequence diagram that models the interactions between these five participants. A participant in any interaction can be an *agent* or an *actor*. The events being exchanged can be one of three types: *messages* between agents within the system (e.g., *refund request*), *percepts* between actors and agents (e.g., *deposit confirmation*) and *actions* between agents and actors (e.g., *deposit amount*).

Detailed design phase

In the detailed design phase, each agent in the system is specified and modelled in detail to allow it to fulfil its responsibilities (i.e., its behaviours and interactions) according to the system-overview diagram. Each agent is modelled in its own *agent-overview diagram* encapsulating the required design entities for realising its desires. Each agent is designed and detailed in terms of events, belief sets and plans to realise the specifications of the sought system.

Plans in Prometheus are *sequences of steps* that assist in realising assigned goals. Each plan must have a single trigger that initiates its execution (signified by a dashed arrow from the event to

[1]The provided goal overview is a possible design and it does not necessarily represent the best one

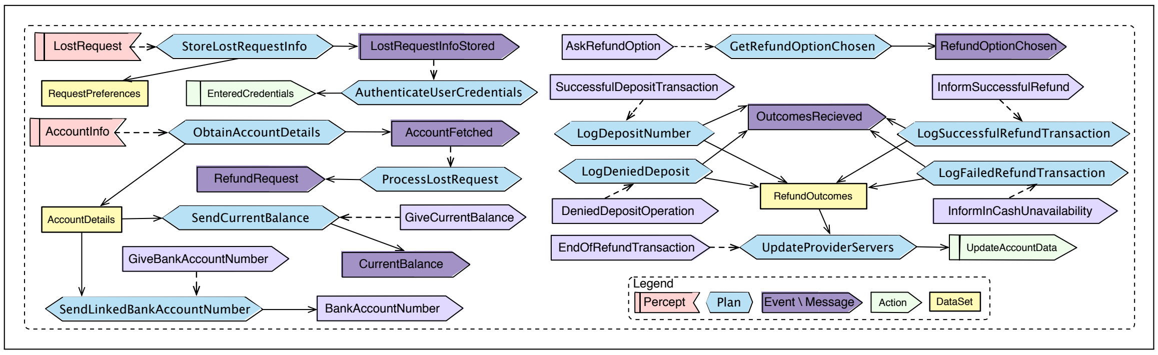


Figure 2.8: Ticketing agent-overview diagram

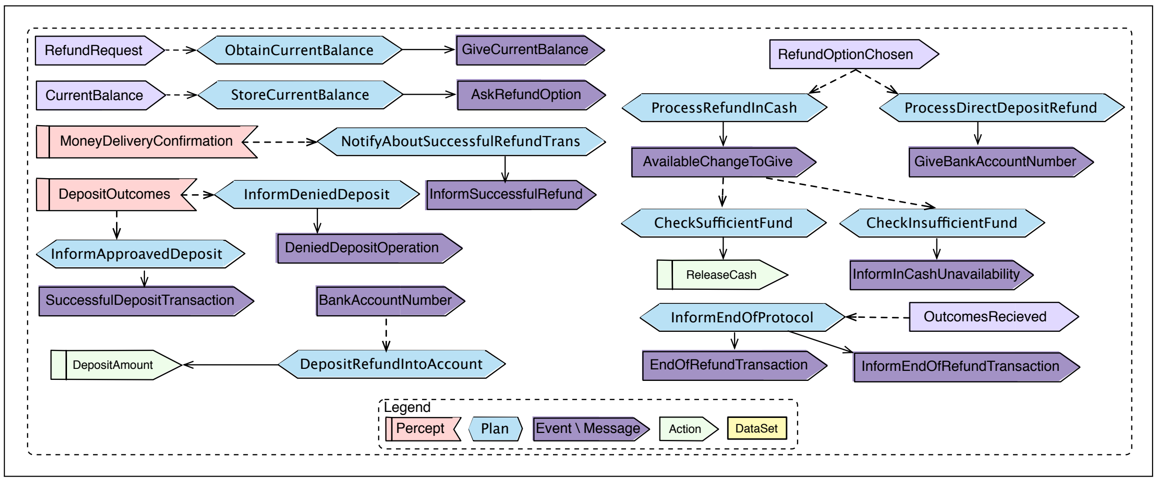


Figure 2.9: Payment manager agent-overview diagram

the plan), and can have multiple inputs to process (signified by a solid arrow from the event to the plan). Messages, events and percepts can act as inputs to a plan. A plan may post internal events (occurring within the same agent) to trigger other plans, it may post actions to interact with the environment or it may send messages to communicate with other agents (signified by solid arrows from the plan to the event). Further, each plan has a descriptor that allows designers to specify a set of properties including *goals* to be achieved by the plan.

Figures 2.8, 2.9 and 2.10 illustrate the detailed designs of the three agents—the ticketing agent, the payment manager agent and the GUI agent—in the transport-ticketing smart kiosk system with respect to the lost-card-request scenario shown in Figure 2.5. The figures show the mapping of the scenario for the detailed design of each participant. The ticketing agent is initiated upon the receipt of the LostCardRequest percept (the trigger of the scenario) from the environment (the touch screen actor). The percept triggers the StoreLostRequestInfo plan that logs the request

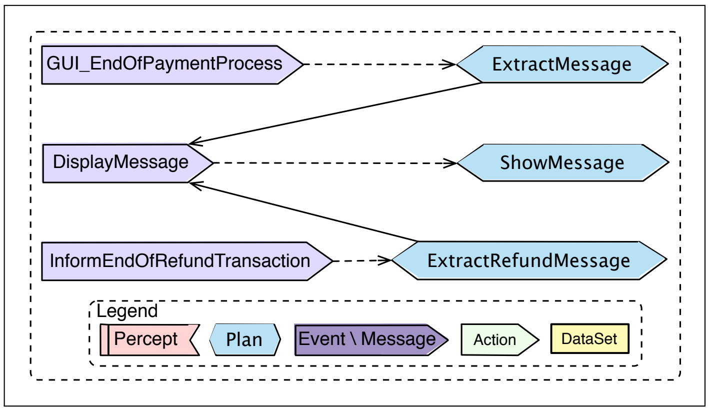


Figure 2.10: GUI agent-overview diagram

information in the RequestPrefernaces dataset and posts an internal event labelled LostRequestInfoStored. This plan, through its descriptor, is assigned the ExtractRequestInfo goal (Step 2 of the scenario). The internal event LostRequestInfoStored is handled by the AuthenticateUserCredentials plan anticipated to realise the VerifyCredentials goal (Step 3 of the scenario). The plan then posts the EnteredCredentials action to the environment (Step 4 of the scenario). According to the scenario in Figure 2.5, the ticketing agent should be idle until it receives the AccountInfo percept. The AccountInfo percept triggers the ObtainAccountDetails plan, which is responsible for realising the GetAccountInfo goal (Step 5 of the scenario). The plan then posts the AccountFetched event to indicate the realisation of the GetAccountInfo goal. That internal event is handled by the ProcessLostRequest plan that realises the RefundOldCard goal by posting the RefundRequest message to the payment manager agent. This message triggers the protocol in Figure 2.7.

In the refund protocol in Figure 2.7, each agent is designed to fulfil its role in the protocol faithfully. The protocol is triggered by the sending of the RefundRequest message by the ticketing agent (see Figure 2.8). The payment manager agent in Figure 2.9 handles the RefundRequest message by the ObtainCurrentBalance plan, and the same plan sends the next message in the protocol GiveCurrentBalance, to ensure the sequence flow that is enforced by the protocol. By following the flow of the events in the designs of Figures 2.8, 2.9 and 2.10, it can be observed that the agents are faithfully implementing the *DepositAmountIntoAccount* option in the protocol.

Tool Support-PDT

Prometheus has a design tool known as PDT [Padgham et al. 2005a], to support its methodology. This tool provides designers with many features for producing well-documented design artefacts for the intended system. Designers can describe the design units on the diagrams using the design descriptors [Padgham et al. 2007]. For example, each plan in the agent-overview diagram has a descriptor. This descriptor allows designers to detail the plan in terms of its context condition, the goals to be realised and an overall description of its purpose.

The tool also has a code-generation feature that transforms the detailed design to a skeleton implementation code in the JACK programming language [Busetta et al. 1999], which creates a good starting point for developers. Jayatilleke et al. [Jayatilleke et al. 2006] extended the tool using the CAFnE toolkit for generating an executable code, rather than a skeleton. This toolkit requires more information about the models to use in the generation of the code. Thus, Prometheus supports only the generation of skeleton JACK codes from the PDT design models. Additionally, there have been some proposals for targeting other implementation platforms, such as Jadex [Sudeikat et al. 2004] and 3APL [Jayatilleke 2007].

Given the above, it is valuable to ensure the validity of design models before the code-generation phase. In other words, the ensuring of consistent and correct design models implies a correct skeleton code for the system-to-be. Although there have been tools for run-time debugging and testing based on PDT models [Padgham et al. 2005b; 2013], there is little support for examining the validity of the Prometheus-based design models.

2.5 Other AOSE Methodologies

This section briefly explains six other methodologies than Prometheus to show that they share similar notions about specifying requirements and interaction protocols. This is not intended to serve as a comprehensive list of methodologies. Such a comparison can be found in [Gómez-Sanz and Fuentes-Fernández 2015].

These six methodologies were chosen based on the approach adopted in Dam and Winikoff [Dam and Winikoff 2013]. Dam and Winikoff adopted a multi-stage selection approach, in which the set of methodologies was reduced by applying the following three criteria:

- *Documentation:* the selected methodology should be well established and well described.

- *Tool support:* a methodology that is supported by a computer-aided software engineering tool has more value than one without such support.
- *Maturity:* the selected methodology has been continuously improved and should be well recognised by the agent community.

Dam and Winikoff [Dam and Winikoff 2013] stated that seven methodologies out of the AOSE methodologies in the literature met these three criteria. We have included all of them, except ADEM ([Trencanský and Cervenka 2005], Section 7.2]) since it provides a framework rather than a methodology. Such frameworks allow software engineers to build customised software engineering methodologies and, hence, the resultant methodologies may vary in terms of artefacts produced. Consequently, such frameworks do not fit the context of this study. We have also included ROADMAP, even though it is weak on the maturity criterion, since it represents an extension of Gaia, and supports similar notions to Prometheus.

Tropos

Tropos has five phases [Bresciani et al. 2004]: early requirements, late requirements, architectural design, detailed design and implementation. In the early requirement phase, the system stakeholders are identified as well as the system's objectives, which reveal the *system's actors* and *goals*. Then, the obligation of the system towards its environment is determined as the main activity of the late-requirement phase. Tropos adopts the i^* organisational modelling framework in its modelling requirements [Bresciani et al. 2004] since the i^* framework supports the notions of goals, actors and their dependencies [Yu 1995]. Even though Tropos does not have the concept of scenarios as part of its requirement-specification process, the application of use cases has been proposed in its extended version: the Secure Tropos methodology [Alam et al. 2015].

Unlike in Prometheus, the communication protocols definition activity does not take place in the architectural design phase of the Tropos methodology. The interaction protocols are specified as part of the capability-modelling activity in the detailed design phase. In Tropos, AUML-sequence diagrams are used to define the communication protocols between agents [Bresciani et al. 2004].

The Tropos methodology has many tools to support the methodology process [Morandini et al. 2007]. One of these tools is Tropos Agent-Oriented Modelling (TAOM4e), which enables the designer to model the intended agent-based system. This tool has a code-generation feature that takes a detailed design and provides a skeleton code in the JADE agent programming language. The tool proposes many generators to facilitate the production of a skeleton code that targets the JADE and Jadex agent platforms based on the detailed design artefacts or on the goal model. Tropos provides designers with run-time testing ability through a tool known as eCAT [Nguyen et al. 2008]. However, it is important to ensure conformity of the design models with the specifications of the system-to-be to guarantee a correct skeleton code.

INGENIAS

The INGENIAS methodology enables agent designers to develop an agent-based system through its five viewpoints [Pavón et al. 2005]: organisation, agent, goals/tasks, interactions and environment. The methodology promotes a number of abstrac concepts, such as agent, goal and mental state. The requirement specification process takes place under the first three viewpoints. During the organisational viewpoint, the goals of the intended system are defined. Further, the tasks that the agents need to execute to achieve the desired goals are specified in this viewpoint. Under the goals/tasks viewpoint, the goals and tasks are decomposed and refined further. The tasks are to be defined in terms of what they are, why they are required, their input, their output and the goals that they are to achieve. Tasks are similar to scenarios in Prometheus.

The definition of interaction protocols in this methodology is part of the interaction viewpoint. Although the methodology has its notations (Grasia[2]) for modelling protocols, it accepts AUML-sequence diagrams to model the interactions [Pavón and Gómez-Sanz 2003].

The IDK is an integrated development environment that supports the methodologie's development life-cycle [Gómez-Sanz et al. 2008]. Along with the aid the tool provides in documenting the design, the IDK provides a code-generation capability that transforms the system's meta-models into an implementation code targeting the JADE platform [Gómez-Sanz and Pavón 2005]. Further, the tool supports the debugging process through the ACLAnalyser tool [Botía et al. 2004].

[2] http://grasia.fdi.ucm.es

MaSE

The MaSE methodology allows for the development of agent-based systems through two main phases: analysis and design [Deloach 2004]. Each phase has steps that result in different artefacts. In this thesis, we are concerned with the analysis phase, whereby the system goals and roles are identified. This phase has three steps: (1) capturing goals, whereby the goal hierarchy is generated; (2) applying use cases, whereby the use cases and sequence diagrams are constructed; and (3) refining roles, whereby the concurrent tasks and roles are modelled. As the first step, the designer needs to identify what the intended system wants to achieve (the system's goal). Then, the system's roles and their tasks are specified in the form of use cases that define the system's desired behaviour. Use cases are elicited from the context of the intended system and are designated as positive and negative. The positive use cases state the normal behaviour of the system, while the negative ones describe behaviour that should not happen. These use cases are then converted into sequence diagrams to visualise the flow of events between different roles. Finally, a transformation of the goal hierarchy and the use cases via sequence diagrams into roles and their associated tasks takes place. Similar to Prometheus, MaSE uses a goal hierarchy to model the goals of the intended system. In MaSE, use cases are specified using sequence diagrams that capture the flow of steps, which provide a richer representation than the scenarios of Prometheus.

In the MaSE methodology, interaction protocols are known as *conversations* between agent classes, following Barbuceanu and Fox [Barbuceanu and Fox 1995]. The construction of conversations takes place in the design phase of the methodology. Unlike Prometheus, MaSE does not use AUML for modelling conversations. Rather, it uses diagrams that are called *communication class diagrams*. A communication class diagram is a set of finite state machine (FSM) diagrams [DeLoach 1999]. Each FSM models one side of the specified conversation. Given this, a conversation class diagram that models a conversation between two participants needs to have two FSMs. Despite MaSE's adoption of FSMs, it still supports the same inter-agent interactions as Prometheus. The conversations model the interactions between multiple participants, as in Prometheus. In addition, FSMs have the same ability as the AUML-sequence diagram constructs to control the flow of communication.

MaSE provides the agent engineers with AgentTool [DeLoach 2001], a graphical modelling environment. The tool supports several functions apart from the modelling, including conversation verification [Lacey and DeLoach 2000] and code generation.

Gaia

The Gaia methodology enables agent designers to analyse and design an agent-based system using two main phases: the analysis and design phases [Wooldridge et al. 2000, Zambonelli et al. 2003]. In the analysis phase, the designers elicit all the possible entities of the intended system by using concepts abstracted from the requirement statements. Gaia models requirements by using four models: (1) an environmental model, (2) a preliminary role, (3) an interaction model and (4) organisational rules. In this thesis, we are concerned with the second (role) and third (interaction) models.

Roles are identified by four attributes: (1) their permissions and rights, (2) their responsibilities or functionality, (3) their activities and (4) their protocols. Responsibilities are similar to scenarios in that they focus on the functions to be executed by agents. Interaction models capture the communication between agents in the system. Gaia adopts its notations from the Fusion object-oriented methodology [Coleman et al. 1994]. Even though Gaia uses other notations than AUML for modelling interaction protocols, the integration of AUML with its methodology has been recommended [Cernuzzi and Zambonelli 2004].

Gaia for Eclipse (Gaia4E) is a tool used as part of the Gaia methodology to help document the analysis and design models of an agent-based system [Cernuzzi and Zambonelli 2009]. Gaia4E provides designers with the necessary modelling capabilities. For example, it does not support code generation—nor propagation—as the phases in the tool are separate from each other.

ROADMAP

The ROADMAP methodology extends the Gaia methodology for developing open systems [Juan et al. 2002]. One of Gaia's weaknesses is the lack of support for the goal notion in the modelling process. Conversely, the ROADMAP methodology overcomes this issue by introducing a number of extensions. First, it introduces an initial phase that concerns the requirement elicitation process via the use cases. Second, the methodology includes the definition of local, social goals as part of the role modelling process.

ROADMAP treats interaction protocols as reusable message patterns that can be manipulated

	Goals	Use Cases / Scenarios	Other notations and models
Prometheus	✔	✔	role overview diagram
Tropos	✔	✘	actor diagram and rationale diagram
MaSE	✔	✔	sequence diagrams and concurrent task model
INGENIAS	✔	✔	organisation model and task model
Gaia	✘	✘	environmental, role, protocol and interaction models
ROADMAP	✔	✔	environmental, knowledge, role, protocol and interaction models
PASSI	✘	✔	UML packages, sequence diagrams and activity diagrams

Table 2.2: Concepts adopted when using the seven AOSE methodologies to specify requirements

without affecting the agent services that underpin them [Juan et al. 2002].

PASSI

The PASSI is a step-by-step methodology that guides designers throughout the life-cycle of the development. The methodology has five process components, referred to as models: (1) the system requirements model, (2) the agent society model, (3) the agent-implementation model, (4) the code model and (5) the deployment model. Each model comprises a number of phases [Cossentino 2005]. This thesis is concerned with the first two models: (1) the system requirements model and (2) the agent society model.

The system requirements model consists of four phases [Cossentino 2005]: (1) the domain requirement description, (2) agent identification, (3) role identification and (4) task specification. In the domain requirement description, the functions of the system are identified through use-case diagrams. The responsibility of each agent is attributed by the agent identification in the form of UML packages; then, the responsibilities of each agent, based on the role-specific scenarios, are explored via sequence diagrams. In the task specification phase, activity diagrams are used to specify the capabilities of each agent.

In the agent society model, the construction process of the social aspect of the intended system takes place. The process of modelling such interactions involves three steps [Cossentino 2005]: (1) ontology description, (2) role description and (3) protocol description. Similar to Prometheus, PASSI uses AUML-sequence diagrams to define the interaction protocols in the intended system. The PASSI Toolkit (PTK) is an add-in for the commercial Rational Rose UML-

	AUML Sequence Diagram	Other notations and models
Prometheus	✔	-
Tropos	✔	FIPA-ACL
MaSE	✘	Finite State Machines
INGENIAS	✔	Grasia
Gaia	✘	Interaction Models
ROADMAP	✘	Interaction Models
PASSI	✔	-

Table 2.3: Concepts adopted when using the seven AOSE methodologies to model interaction protocols

based tool and it supports the PASSI methodology [Chella et al. 2004]. Although this add-in enables designers to construct diagrams from scratch, it also automatically generates other full or partial diagrams to be completed by designers. The tool can produce comprehensive reports on designs, including diagrams and their descriptions of the intended system. This tool does not support information propagation between the methodology phases, though it does offer a code-generation facility.

Discussion

This section has highlighted six other AOSE methodologies than Prometheus to show that they each share similar notions, especially in terms of specifying requirements and modelling protocols. Even though each methodology has its unique development process, each bears in common the inclusion of the core development activities: analysis, design, implementation and testing. Further, all methodologies with the exception of ROADMAP are supported by tools to assist designers in implementing and documenting the development process.

In the agent-oriented software engineering methodologies discussed in this chapter, the requirement specifications include scenarios ([Winikoff and Padgham 2013], Section 4) that are instances of the desired execution behaviour and goals (see Table 2.2). According to Table 2.3, four methodologies use AUML-sequence diagrams for modelling protocols. Although MaSE does not adopt AUML notation, it employs the same communication approaches as the other four methodologies. MaSE views the communication protocol as a means of governing message-passing between multiple participants. However, Gaia and ROADMAP do not view interactions

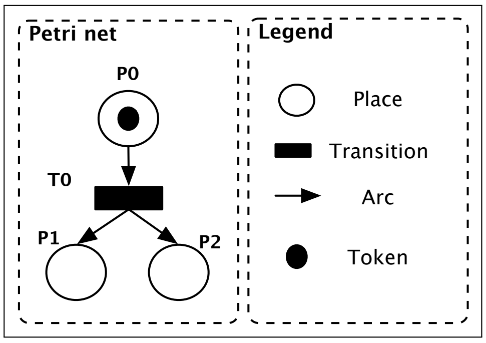


Figure 2.11: *Place/transition* Petri net

as a sequences of messages being exchanged in a certain order. Instead, interaction protocols are treated as a means for defining interaction patterns. Hence, at run time, the same protocol may result in a number of message interchanges.

2.6 Petri Nets

In our approach, Petri nets were used both as a formal representation of the specification of the system and to extract all possible behaviour runs from the agent-behaviour models (see Chapter 5). We provide a brief introduction to Petri nets in this section.

Petri nets (named after Carl Adam Petri) are one of the most commonly used formalisms for describing and modelling concurrency, communications and synchronisation. Petri nets have been used in various ways for analysing, modelling and implementing agent-based systems [Purvis and Cranefield 1996, Mazouzi et al. 2002, Cost et al. 2000, Köhler et al. 2001]. Various types of Petri nets have been proposed to overcome the limitations of the initial formalism, including *coloured Petri nets* [Jensen 2013], *timed Petri nets* [Wang 2012], *object Petri nets* [Valk 2004] and *place/transition Petri nets* [Desel and Reisig 1998]. This thesis focuses on *place/transition Petri nets*, which are formally defined in Definition 1.

Definition 1. A *marked Petri net* is a tuple where $M = \{P, T, I, O, \mu\}$, in which P is a finite set of places, T is a finite set of transitions, such as $P \cap T = \emptyset$, I is an input function, O an output function, and μ is the marking of the Petri Net defined as an n-vector, $\mu = \{\mu_1, \mu_2, \mu_3, ..., \mu_n\}$, in which $n = |P|$ and each $\mu_i \in N, i = 0, 1$.

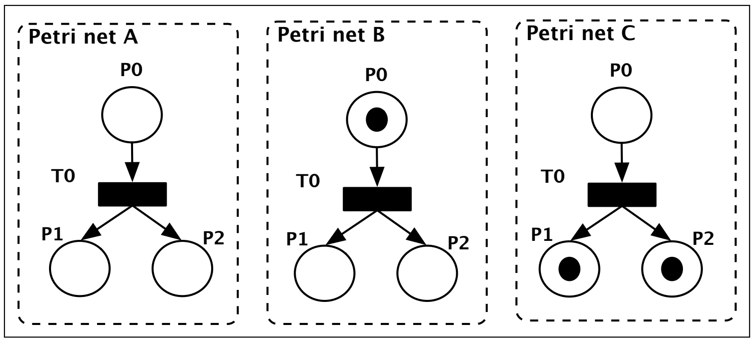
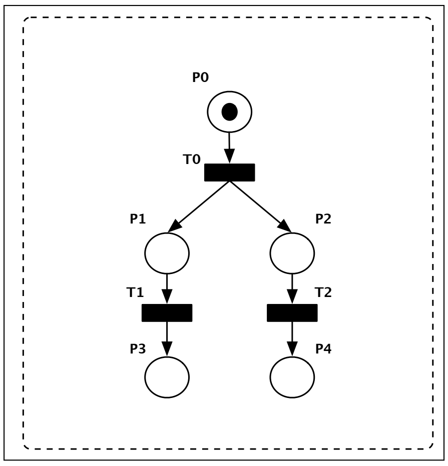


Figure 2.12: Petri net firing mechanism

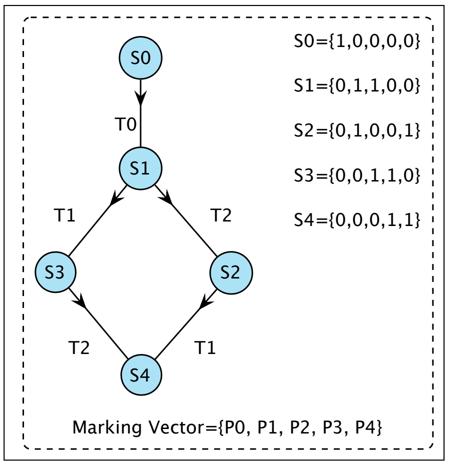
Place/transition Petri nets are one of the most prominent types of Petri nets. In this kind of net (see Figure 2.11), places (denoted graphically by circles) are connected to transitions (denoted graphically by rectangles) via arcs (donated graphically by arrows). Arcs represent the *flow relation* between places and transition. These flow relations are only between places and transitions or vice versa, and not between places or between transitions alone. Each place in the net should have a name and may contain *tokens* (denoted graphically by filled circles). Markings are vectors that show the distribution of tokens across the places of a given Petri net. Although a place in this kind of net can contain a *finite number* of tokens, the approach of this thesis restricts this number to *zero* or *one* token (refer to Definition 2). These tokens determine the execution (*firing*) of the Petri nets. Each transition in a Petri net should have a name, a set of input places and a set of output places. Places with arcs that proceed to the transition are input places, while those with arcs that emerge from the transition are output places. For example, in Figure 2.11, place *P0* is the input place for transition *T0*, while *P1* and *P2* are its output places.

Execution rules for Petri nets

The execution of Petri nets is determined by the *marking* of the net. Briefly, a Petri net executes by *firing* transitions and such an operation is contingent on having the transitions *enabled* first. A transition is enabled if—and only if—all of its input places have tokens. The consequence of firing a transition is the removal of one token from each input place of the transition and the placement of a token on each output place. For example, the Petri net in Figure 2.12 (A) will not execute since *T0* is not enabled (i.e., the input place *P0* does not have tokens). However, the transition in Figure 2.12 (B) will fire since *P0* has a token and, hence, the transition is enabled. Figure 2.12 (C) shows the result of the firing of transition *T0* in Figure 2.12 (B). As is shown, a



(a) Petri net



(b) Reachability Graph

Figure 2.13: Petri net and its reachability Graph

token was removed from *P0* and a token was placed on both *P1* and *P2*.

Analysis of Petri nets

This thesis uses Petri nets as a means of representing system specifications and extracting all possible behaviour runs from the agent-behaviour models. Given this, it is vital to analyse the Petri net that models the agent-behaviour models, especially, for the purpose of extracting the behaviour runs. Petri nets are analysed through their behavioural properties: *safeness*, *liveness* and *reachability* ([Peterson 1981], page 79). *Reachability* is one of the basic properties that emerging from the initial marking of the Petri net. In fact, it is the fundamental property to be considered when studying the behaviour of any system [Murata 1989]. The reachability property calculates the reachable markings of a Petri net, given its initial marking. In other words, it exposes the state of the Petri net (i.e., the distribution of tokens across the net) with every transition. Several techniques have been developed to assist with analysing the properties, as mentioned above, including the *reachability graph* [Murata 1989].

The *reachability graph* of a Petri net depicts a transitional relation that defines the state and transitions of the net. Each state within the reachability graph is, in fact, a *marking vector* that shows the distribution of tokens across the places of the Petri net. Figure 2.13b shows the reachability graph for the Petri net indicated in Figure 2.13a. When the reachability graph is in state S0, this represents the marking $\{1,0,0,0,0\}$. Using the legend at the bottom of Figure 2.13b,

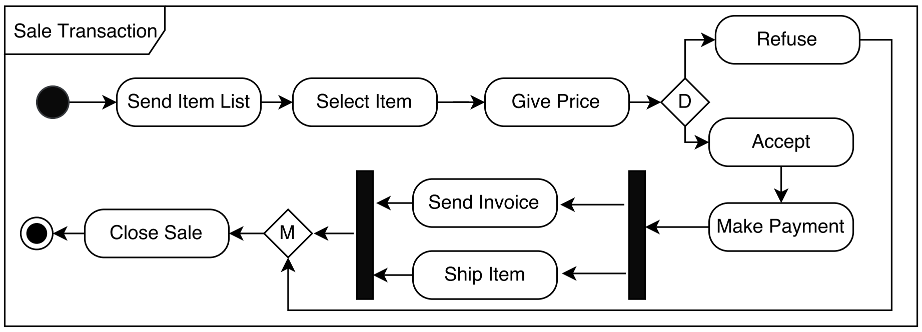


Figure 2.14: Activity diagram for a sale transaction process

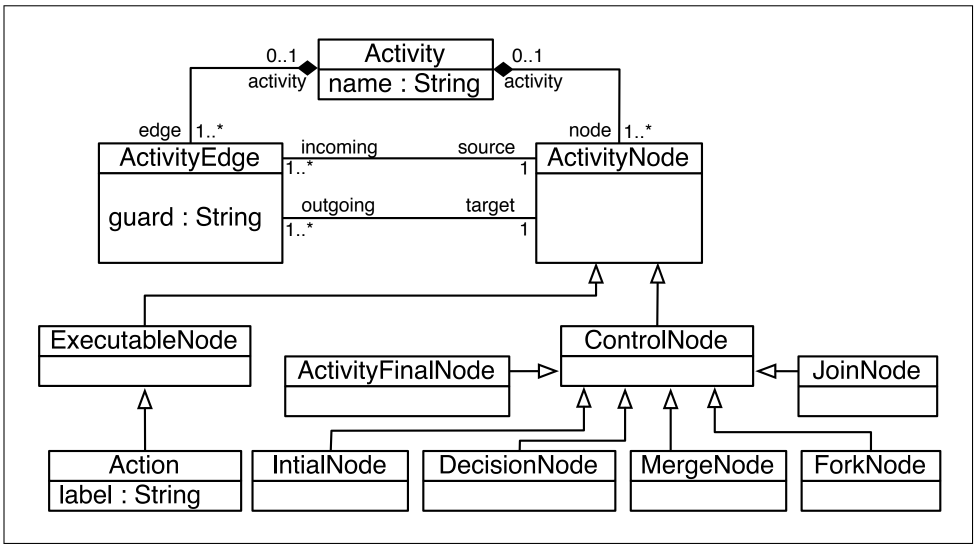


Figure 2.15: Activity diagrams meta-model

this means that a token is on the Petri net place 'Start', because the first slot on the marking is one, while the remainder is zero.

2.7 UML-activity Diagrams

In this work, we propose using UML-activity diagrams as more structured representations of requirement specifications to complement the process of the Prometheus agent design methodology (see Chapter 4). Thus, we briefly explain the UML-activity diagrams and the adopted notations of our approach in this section. Activity diagrams are a type of behaviour model that describes the dynamic aspects of a given system [Object Management Group 2011]. They visualise processes (using a sequence of steps) and model the work and data flows of the system. The activity diagram in Figure 2.14 demonstrates the process that takes place in a sale transaction. After the item list is

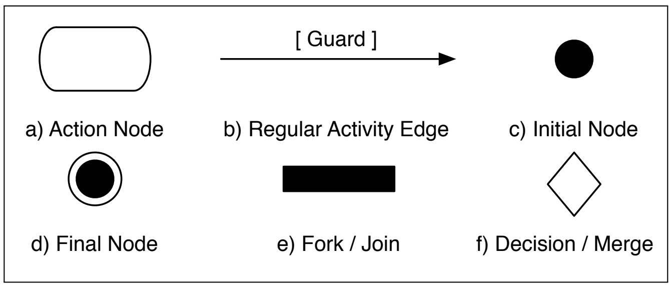


Figure 2.16: Adopted UML-activity diagram notations

sent and an item is selected, the price will be either accepted or refused (via a decision point). If the price is refused, the sale-transaction process will flow to the merge point, allowing the 'Close Sale' action to be performed. However, when a price is accepted, the flow will perform the 'Make Payment' action and the item will be shipped before closing the sale transaction. Note that, according to the activity diagram, this sale-transaction process allows for the shipping of an order before the sending of an invoice (i.e., for parallel action nodes).

UML provides designers with a range of graphical notations for modelling activity diagrams including activity nodes and activity edges. The creation of these diagrams must conform to the UML-activity meta-model to ensure the diagrams' semantics. In fact, this meta-model provides an enormous number of entities for enhancing the semantics of activity diagrams. Figure 2.15 shows a subset of the activity diagram meta-model that defines the entities considered in the approach described in Chapter 4. As Figure 2.16 illustrates, six entities out of the UML activity diagram notation are adopted as follows:

1. *Action Node*: the primary atomic node in the activity diagrams (see Figure 2.16(a)). It is notated by a rectangular shape with rounded edges.

2. *Regular Activity Edge*: a directed connection that shows the control flow between the different actions of an activity diagram (see Figure 2.16(b)). These edges can optionally have conditions (*guards*) to constrain their flow.

3. *Initial Node*: a control node that initiates the execution of an activity/scenario (see Figure 2.16(c)). We restrict the activity diagrams to have only one initial node.

4. *Final Node*: a control node that shows the end of an activity/scenario (see Figure 2.16(d)).

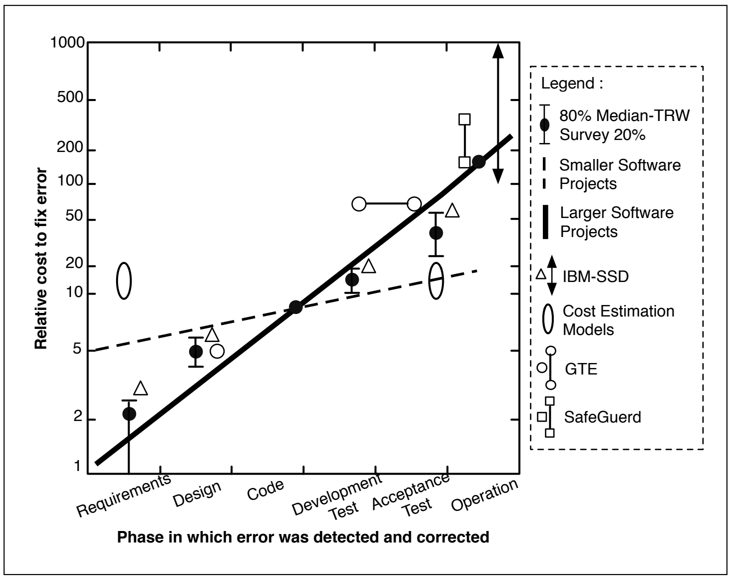


Figure 2.17: Cost of fixing bugs based on time of detection

5. *Fork / Join*: a control node that splits the execution flow into multiple threads to be executed independently, and then synchronises them (see Figure 2.16(e)).

6. *Decision / Merge*: a control node that splits the execution of an activity into multiple alternate flows with only one flow to be executed based on the *guards* for the outgoing activity edges of the decision point (see Figure 2.16(f)).

2.8 Software Verification and Validation

The development of software consists of a number of activities including analysis of requirements, design, implementation, testing, deployment and maintenance ([L. Pfleeger 2010], page 24). These development activities are performed by software engineers (i.e., humans). Hence, they are error-prone activities. These errors may result in undetected faults in the final product and can consequently lead to significant failures in its behaviour. Thus, software verification and validation are necessary, since its objective is to detect and resolve defects early in the development life-cycle. In fact, it has long been established in software engineering that the early detection and resolution of software defects saves time and money, especially for large projects [Boehm and Papaccio 1988, Boehm et al. 1981]. Figure 2.17 [Boehm 1984] shows that fixing defects in large projects early in the life-cycle can save up to a factor of 100 of the cost, whereas in small projects

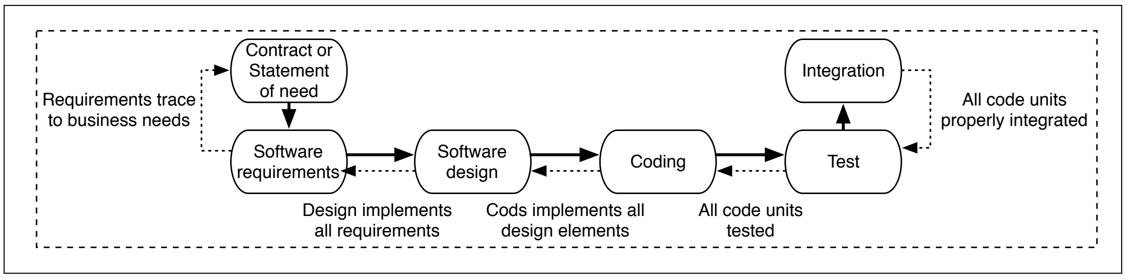

Contract or Statement of need
Requirements trace to business needs
Software requirements
Software design
Coding
Test
Integration
All code units properly integrated
Design implements all requirements
Cods implements all design elements
All code units tested

Figure 2.18: Software verification and validation (static category)

it can save a factor of four of the cost. Additionally, software verification improves the quality of software products, since it ensures consistent and defect-free development. This section briefly introduces some background knowledge about software verification.

Software verification is a software engineering processes that examine whether a software and its associated products fulfil their specifications and whether it behaves as intended. The recent IEEE Standard for System and Software Verification and Validation defines 'verification' and 'validation' as follows ([IEEE], page 11):

- Verification: the process of assessing the conformance of a software and its related products, throughout its life-cycle activities, with the requirements of the intended system.

- Validation: the process of examining software and its associated products to determine whether or not they satisfy the requirements at the end of each activity in the development life-cycle, and also of checking that the software behaves as intended.

Software verification is commonly interpreted as the application of formal approaches to software programs (e.g., model checking [Clarke et al. 1999]). In fact, there have been a number of techniques proposed to help software engineers with the verification and validation processes. These techniques range from formal techniques [D'Silva et al. 2008] to semi-formal techniques [Cordeiro et al. 2009, Kececi et al. 2002, Stavely 1999] to informal techniques, such as desk-checking ([Beizer 2003], Chapter 3) and inspections and walkthroughs ([Schach 2008], pages 148-153). Software verification and validation techniques fall into two categories: *static techniques* and *dynamic techniques* ([Ghezzi et al. 2003], pages 269-274). The static category involves analysing the documentation of the *system-of-interest* without executing it (see Figure 2.18), while

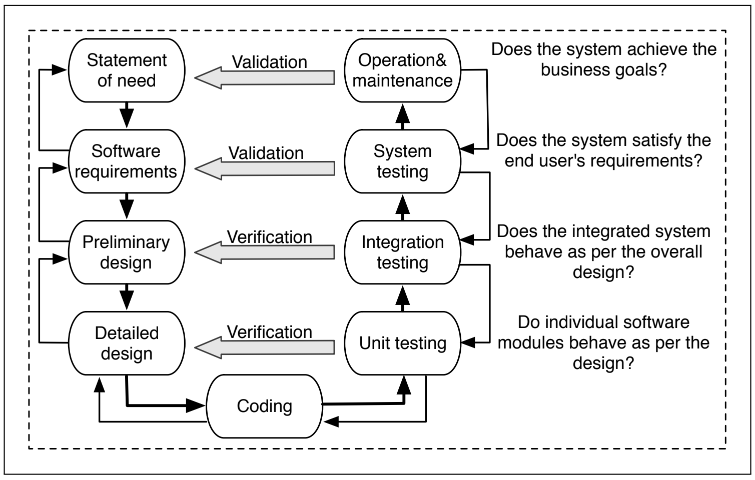

Statement of need
Validation
Operation& maintenance
Does the system achieve the business goals?
Software requirements
Validation
System testing
Does the system satisfy the end user's requirements?
Preliminary design
Verification
Integration testing
Does the integrated system behave as per the overall design?
Detailed design
Verification
Unit testing
Do individual software modules behave as per the design?
Coding

Figure 2.19: Software verification and validation (dynamic category)

the dynamic approaches require the execution of the system-of-interest since they experiment with its behaviours (see Figure 2.19).

As Figure 2.19 shows, the dynamic software verification and validation techniques are concerned with *testing* the behaviour of the system-of-interest and ensuring its conformance with respect to the part of the system under test (the so-called *target*). The testing process encompasses different levels and begins with *unit testing* (also known as *module testing*) ([Bourque et al. 2014], Chapter 4). Unit testing ensures that individual software modules behave as per the design. The next level is *integration testing*, which integrates all the modules into subsystems and tests whether the integrated system behaves as per the overall design. Then, the integrated system is validated as to whether it meets the functional and non-functional requirements at the *system testing* level, which begins once all components have been successfully integrated and tested.

Conversely, as Figure 2.18 shows, the static approach analyses the development artefacts of the system-of-interest without executing the system. For example, following the *inspection* technique, the *inspectors* walk through the design artefacts of the system-of-interest and ensure that they correctly implement all requirements. The work proposed in this thesis focuses on verification, especially on the *static verification of software designs*. Validation, by contrast, falls beyond the scope of this thesis; it is often relevant to requirements and to approaches for checking their completeness, feasibility and testability ([Sommerville and Sawyer 1997], pages 189-214).

2.9 Related Work

This section discusses existing techniques for verifying software designs, in the context of general software engineering and agent-oriented paradigms. We cover software verification in this broader area (i.e., general software engineering) to show how the verification of agent-based systems fits into this broader context. Section 2.9.1 serves as an introduction to the criteria for which a verification process of software designs should aim. It also provides an overview of the two prominent verification technique categories: manual and automated.

Techniques for detecting defects and inconsistencies in object-oriented designs using UML notation have been studied more than the agent-oriented paradigm. Existing approaches to verifying object-oriented designs are discussed in Section 2.9.2. There have also been approaches developed for assuring the correctness of agent-based designs as discussed in Section 2.9.3.

2.9.1 Software Design Verification

As mentioned in Section 2.8, the principal objective of statically verifying software designs is to detect and resolve defects early in the development life-cycle and, hence, to save development costs. Boehm [Boehm 1984] has identified four criteria for the verification process to look for in the system-of-interest: *completeness, consistency, feasibility* and *testability*. Since the main emphasis of this thesis is to check agent designs against specifications, the testability and feasibility criteria will be omitted. This is because they do not strongly relate to the design of the system of interest. Testability is more relevant to requirements than design. The criterion of testability is to ensure that the system of interest will meet its specifications through ensuring that the requirements are specific and unambiguous [Boehm 1984]. The feasibility criterion is about verifying that the system of interest can be developed such that it satisfies its functional and performance requirements; thus, it is concerned with the management and financial aspects of the intended system [Boehm 1984]. Given this, we only consider the first two criteria: completeness and consistency. However, we redefine the consistency criterion to suite our approach, as we do not ensure consistency within the same levels of artefacts; for example, requirements are consistent with each other and design items are consistent with each other, whereas consistency is not considered across boundaries. We check that a design is consistent with its specification, which means that the design is correct with respect to its specification. Thus, the consistency criterion of our approach is, in fact, a *correctness* criterion.

Completeness is about ensuring that the system of interest does not neglect any part of its specifications. In the context of this thesis, completeness means that the design fully *covers* the specifications of the system of interest. Revisiting the running example in Section 2.4, one of the functionalities of the system is to allow customers to process a refund request for a lost card. This process is represented as a goal that needs to be achieved. A design that does not realise this goal is considered incomplete.

Correctness is about ensuring that there is no conflict among the design artefacts of the system of interest (i.e., of the agent-behaviour models with their specifications). Further, this criterion considers traceability of the artefacts as a way of acquiring a consistent system. Traceability is the ability to link all components in later phases of the development life-cycle to earlier phases. Additionally, assuring that the design of the system of interest will result in correct behaviour with respect to a part of the specifications is another approach for meeting this criterion (i.e., of consistent behaviour). For example, the scenario ‘S1’ states that goal ‘G1’ happens first and then goal ‘G2’. A design that has the plan to realise ‘G2’ before realising ‘G1’ is inconsistent with respect to scenario ‘S1’.

2.9.2 Correctness Assurance in Object-Oriented Designs

In the object-oriented paradigm, techniques for detecting defects at the design stage have been extensively studied compared to the agent-oriented paradigm. These techniques range from manual techniques, such as reading techniques including inspections and walkthroughs [Fagan 2001, Travassos et al. 1999], to automated techniques [Usman et al. 2008].

To this end, the main focus of the existing automated software verification approaches is to analyse the correctness of software programs (i.e., their source codes) using methods such as model checking, bounded model checking and abstract static analysis [D’Silva et al. 2008], with little attention to design documents. One of the reasons behind this lack of attention to design documents is that design notations are inexpressive and imprecise and, hence, they cannot be verified. However, there have been efforts in this direction (i.e., towards verifying design artefacts) through the creation of new modelling notations, such as Alloy notations for designing object-oriented software [Jackson 2002]. Additionally, there have been attempts to constrain existing modelling notations, such as the de-facto standard modelling notation UML, so that they can be automatically verified [Rumbaugh et al. 2004]. This thesis focuses on verifying agent-based design models; hence, we do not discuss the automated checking techniques in the object-oriented paradigm because they have been developed specifically for that paradigm. Therefore, they are not applicable to the agent-oriented paradigm, as each paradigm has different design concepts that need to be considered.

Although the manual techniques can be generalised to agent-based systems, there are many different design aspects in the agent-oriented paradigm that must be addressed. In the context of a BDI model of agency, systems are specified in terms of goals that are achievable by agents. These agents are then designed in terms of events, goals, actions, percepts, data sets and plans with their relationships. Manual techniques are costly and suit small projects, while large software projects need automatic aid for their verification. In what follows, we discuss some of these manual techniques.

Manual design verification techniques

As their name indicates, manual techniques are to be employed by humans and do not involve any automation. However, the manual label should not imply a lack of computer-aided software that assists in the verification process [Macdona et al. 1995, Anderson et al. 2003]. Manual techniques are often considered informal since they rely on human subjectivity and not on mathematical formalisms. However, these techniques should be applied using well-structured approaches under formal guidelines, such as the inspection technique [Bush 1990, Parnas and Lawford 2003]. Many techniques have been proposed in this context: (1) the audit technique that assesses the development of products according to the plans, policies, procedures, standards, and guidelines of the intended system ([William 2006], page 26); (2) the desk-checking technique, which comprises an intensive analysis of the software documentation to ensure its correctness, completeness, consistency and clarity [Beizer 2003]; (3) the walkthroughs technique ([Schach 2008], page149); and (4) the inspection technique [Fagan 2001].

Walkthrough and inspection techniques are types of reviews ([Sommerville 2009], page 663) that detect and document faults in the software documentation and ensure their correctness. However, unlike walkthroughs, inspections are formalised processes that use checklists to guide teamwork to reveal defects in the documentation of the intended software design. Inspections are some of the most cost-effective and commonly used manual techniques for detecting defects in software documentations.

Inspections are usually conducted by a team of three members—a moderator, a reader and a recorder—each of whom are responsible for managing the inspection process and leading the team ([Schach 2008], pages150-152). The inspection process begins with the planning phase. At the planning phase, the participants are selected and the materials are prepared. The inspection technique involves five steps: overview, preparation, inspection, reworking and follow-up. The overview comprises an informal meeting where a summary of the design and its documentation including the inspection agenda are distributed to members of the team. In the preparation phase, each member of the inspection team carefully reviews the documentation provided to them in the overview meeting. The moderator then plans and manages the inspection meeting. The moderator presents the system of interest and the inspection process to the inspection team. The inspection team is also provided with the necessary checklists to be used in the inspection. The objective of this phase is to count defects in the system of interest without fixing them. At the end of this phase, the moderator prepares a report that logs the defects identified by the inspection team and that is to be distributed among the team. In the reworking phase, the design team attempts to resolve the defects reported while documenting their responses. During the follow-up phase, the moderator works through the changes made by the designers in the previous phase and ensures that they do not introduce new problems.

Although inspection is an effective technique for revealing defects in software documentations, it is a costly process, as it requires manual investigation by a number of investigators. The inspection process becomes particularly tedious when the system of interest is large. In this case, there is a need to support the inspection process with a computer tool. A number of tools have been developed to improve the inspection process, such as Collaborative Software Inspection (CSI) [Mashayekhi et al. 1993] and Scrutiny [Gintell et al. 1993]. Macdonald *et al.* [MacDonald et al. 1996] designate the areas in which inspections need to be automated as document handling, individual preparation, meeting support and metrics collection. They list and compare five tools that support the inspection process.

2.9.3 Correctness Assurance in Agent-Based Designs

In terms of agent-design methodologies, there has been little research into how to ensure the correctness of agent-design artefacts. There is a large body of work on verifying BDI-agent systems using formal verification, such as model checking (e.g., [Dastani et al. 2010, Bordini et al. 2003; 2006b, Dennis et al. 2012]) and theorem proving (e.g., [Shapiro et al. 2002]). While these approaches can provide a higher level of assurance than the work proposed in this thesis, they assume the existence of either a formal model that abstracts the behaviour of the system or a system implementation. In contrast, the approach propsoed in this thesis aims to detect defects in semi-formal models without requiring source code.

Similarly, researchers have investigated specific methods for the run-time testing of multi-agent programmes [Nguyen et al. 2012; 2007], including BDI agents [Padgham et al. 2013, Miller et al. 2010], assertion-based verification with static analysis [Sudeikat et al. 2006], automatic analysis of agent behaviour through software comprehension [Bosse et al. 2006, Lam and Barber 2005] and run-time debugging of agent interaction [Botía et al. 2004, Padgham et al. 2005b]. Clearly, testing or verifying the implementation of the system of interest provides some level of verification of the corresponding design; however, there is clear value in the ability to detect design defects before implementing a design.

Further, there is an entire body of work in formally verifying interactions between agents in multi-agent open systems [Baldoni et al. 2010; 2009; 2006; 2005]. For example, the ISLANDER tool, which supports the specification of the norms and protocols in electronic institutions, provides a verification module [Esteva et al. 2002]. The ISLANDER tool allows software engineers to define the specification of the sought electronic Institute system formally via a textual language. The verifying module in ISLANDER verifies four properties in the specification of the system: Integrity, liveness, protocol correctness and norm correctness. The tool checks the integrity of the specification of the system by ensuring that each referenced element in the specification is defined. Regarding the liveness property, the tool guarantees that deadlock will not exist in the system. Although the ISLANDER verification module validates the correctness of protocols and norms, these checks are not applicable to the BDI style agent methodologies, such as Prometheus. The module assures that the graph representing the specified protocol is connected. Also, the tool verifies that the protocol is correctly typed as per the backus-naur form (BNF) grammar. Finally, the verification module in ISLANDER tool verifies that agents in the system can perform the actions defined in a norm.

This section provides some insights into existing work that seeks to ensure the correctness and completeness of agent-based models in three areas:

1. Automated consistency and correctness checking
2. Traceability
3. Automatic design propagation mechanisms

(1) Automated consistency and correctness checking:

Despite the fact that most AOSE methodologies offer development environments through their supported tools [Pokahr and Braubach 2009], many do not provide a framework for fully verifying their produced design artefacts. Although some of the AOSE methodologies partially support the verification of agent designs, they do not provide any support for verifying agent-behaviour models against requirements models. In what follows we discuss the methodologies that offer some support for design-time verification.

The Tropos methodology: Tropos provides partial support for requirement verification. It offers two frameworks, each including tool support, for (i) validating formal requirement specifications using the T-Tool [Fuxman et al. 2001] and (ii) reasoning with formal goal models using the GR-Tool [Giorgini et al. 2005].

The T-Tool takes a specification for the system of interest and builds its equivalent automaton, which exhibits all possible execution traces that satisfy the constraints of that specification. The tool then verifies the expected behaviours based on the assertion and possibility properties specified. It also checks the consistency of the specifications by ensuring that there is at least one scenario in the system that respects all the constraints enforced by the requirement specifications.

The GR-Tool enables forward and backward reasoning for goal models. Forward reasoning entails forward propagation of the initial values that have been assigned to some goals in the model to all other goals, according to a set of rules, to determine the satisfiability and dependability of the goals and, consequently, to reveal possible conflicts. Backward reasoning is concerned with finding, through the conducting of a backward search, desired assignments for the goal model when provided with various input values. Although Tropos ensures the validity of the requirement models of the system-to-be, it does not support the verification of the agent-behaviour models (i.e., low-level agent designs) against these requirement models.

The INGENIAS methodology: The IDK [Gómez-Sanz et al. 2008] supports the INGENIAS methodology. The IDK integrates a tool called the ACLAnalyser [Botía et al. 2004] for analysing the interactions between agents at the design phase, when executing the system and logging the run-time interactions [Botía et al. 2006].

The ACLAnalyser tool consists of four components: *the sniffer agent, the relational database, the monitor* and *the analyser*. The sniffer agent is responsible for acquiring a copy of all messages exchanged in the system of interest. Messages are then deposited into a relational database for additional manipulations. The monitor component is in charge of visualising the processing of the sniffer agent. The analyser interacts with the rational database and analyses the logs resulting from the execution of the interactions. In the context of INGENIAS, the ACLAnalyser tool uses the JADE automatic code-generation module to generate templates for agents in the JADE platform, with the aim of compiling and executing these templates. While the agents are being executed, the sniffer agent logs information about the interactions. These logs are then used by the analyser component to report the outcomes of the interactions to the designer.

Even though ACLAnalyser tool gives feedback about the defined interactions at the design phase, it relies on their implementation. It uses the JADE code-generation module and, hence, the agent templates will not be generated if the models are lacking certain information. Although, through the ACLAnalyser tool, the IDK provides valuable feedback about the interactions specified in the design phase, it does not provide any support for verifying agent-behaviour models against requirement models.

The O-MaSE methodology: The O-MaSE methodology offers a development environment for developing agent-based systems through AgentTool III (aT3) [DeLoach and Garcia-Ojeda 2010].

The aT3 tool provides a verification framework that maintains consistency between related models. The check is conducted based on a set of predefined rules that rely on the static relationships between design entities. For example, there is a rule that constrains the goal diagram to the form of a tree and, hence, if a goal node does not appear in a tree format, an error is triggered by the framework. Such checks do not ensure the conformance of agent-behaviour models against any point-of-reference artefacts in the process, including requirement models and interaction protocols.

The Prometheus methodology: Early versions of the PDT [Padgham et al. 2008] offered limited static consistency checking of the design for warnings, such as events that are not handled and messages that should be sent/received according to a protocol. Unlike aT3, this way of checking does not rely on a set of rules that a designer can enable and disable based on their context. On the contrary, it relies purely on the static associations between different design entities. It is worth noting that what we are proposing in this thesis goes beyond these simple static consistency checks, as our proposal considers the dynamic behaviour of a system.

The PASSI methodology: PASSI offers the PASSI Toolkit (PTK) to aid software engineers in developing an agent-based system. Similar to PDT, the PASSI Toolkit (PTK) ensures consistency among the various diagrams by automatically constructing parts of several models and conducting checks on changes made by the designer to verify their correctness with respect to the other parts of the design [Cossentino 2005]. PTK does not support the verification of low-level designs by comparison with top-level artefacts. The work in this thesis, in contrast, assesses the dynamic behaviour of designs.

The Gaia methodology: The Gaia methodology provides agent engineers with a graphical environment for developing agent-based systems, such as the Gaia4E [Cernuzzi and Zambonelli 2009]. This tool does not offer any framework for verifying and validating the agent design artefacts. However, there has been an attempt to transform Gaia role models into business process models with the aim of validating their liveness formulas [Mitakides et al. 2015].

Role models are defined in the analysis phase of the methodology to identify the possible roles agents must fulfil within the intended system. As mentioned in Section 2.5, roles consist of four attributes including responsibilities. Responsibilities can be considered the key attribute, since they define the functions of the system. The responsibilities attribute has two properties: liveness and safety. Liveness properties describe— through a liveness model—the activities an agent must accomplish in specific environmental states. A liveness model connects the activities of an agent with Gaia operators using formulas [Zambonelli et al. 2003].

Mitakidis et al. in [Mitakides et al. 2015] propose an algorithm with tool support—using the Liveness2XPDL tool—to transform the liveness formulas into XML process definition language (XPDL) [Palmer 2009], which can then be converted to business process modelling notations (BPMN). Following this, the BPMN can be simulated using existing simulation platforms, such as Signavio[3]. Such simulation mainly allows for validating the non-functional requirements of the system of interest, such as its delivery time. Designers need to define scenarios for the simulations and each scenario should have its parameters: its role resources, its activity duration and its frequency of role executions. Although the Liveness2XPDL tool fully supports the automatic transformation of liveness formulas into XPDL, the method is limited to role interactions (i.e., to message flows between different activities), since it cannot automatically determine the flow of messages. Moreover, the approach in [Palmer 2009] is more concerned with verifying the analysis models and does not consider the agent-behaviour models of the proposed system.

The MaSE methodology: The AgentTool aids software engineers in documenting the MaSE methodology's activities in terms of design artefacts [DeLoach 2001]. The tool supports the automatic verification of conversations between agents [Lacey and DeLoach 2000]. As mentioned earlier in Section 2.5, MaSE uses FSM diagrams when modelling conversations. The AgentTool then automatically converts these FSM diagrams into the formal modelling language Promela to be checked via the Spin model checker [Holzmann 1997]. AgentTool then graphically visualises the errors detected by Spin.

Although Spin can detect many errors, AgentTool supports the following types: deadlocks,

[3]Web-based process modelling platform (http://www.signavio.com/bpm-academic-initiative).

non-progress loops, unused states, unused messages and mislabelled transitions. Despite the fact that AgentTool adopts a formal approach to verify the conversations, which offers a higher level of assurance, it does not consider the agent-behaviour models in such verifications.

(2) Traceability:

Traceability is helpful not only because it enables designers to maintain a correct design with respect to any changes in the requirements, it also ensures consistency between various design models and their completeness regarding missing design entities.

Castro et al. in [Castro et al. 2002] suggest the adoption of requirement traceability techniques in the agent-oriented paradigm. In particular, they present a way that the AOSE methodologies could embed the general-purpose traceability technique described in [Toranzo 2002]. This traceability approach rests on the inclusion of a meta-model that defines the parameters for the traceability and the reference models and that facilitates the construction of traceability models. In further work, Castro et al. [Castor et al. 2004] integrate the traceability approach mentioned above with the Tropos methodology. In their work, they propose three activities to assist in generating the traceability models—information gathering, information structuring and construction of the traceability matrices—to be integrated with the Tropos phases: early requirements, late requirements, architectural design and detailed design. By adopting this approach, designers can trace the life of a requirement in both forward and backward directions and, hence, are able to ensure the quality of the system of interest. However, the authors do not mention the applicability of their approach to the agent-behaviour models; they only show how it works to address architectural design. Further, the approach does not consider the inconsistencies caused by entities irrelevant to the requirement models.

Cysneiros and Zisman [Filho and Zisman 2008] introduce a rule-based traceability approach grounded in the Prometheus methodology and the JACK platform. The approach automatically generates traceability relations for verifying the models (the agent designs and the JACK code) and checking their completeness by identifying missing elements. Although such an approach is useful in ensuring the completeness of the system of interest, it requires significant implementation compared to the work in this thesis, where we ensure the validity of the design models prior to their implementation.

Thangarajah et al. in [Thangarajah et al. 2011] propose a mechanism that facilitates the traceability between requirements (via scenarios) and other design entities, which is used to support test case generation. The proposed approach extends the existing scenario structure of the Prometheus methodology by three parts: the I/O sequence list, a parameter-based test descriptor and traceability links. The latter part (the traceability links) introduces three relationship links between the scenario steps and four design entities: goals, actions, percept and events. The work of Thangarajah et al. [Thangarajah et al. 2011] is complementary to the approach proposed in this thesis, as it contributes to the consistency assurance of design models.

(3) Automatic Design Propagation Mechanisms:

Another way of ensuring the correctness of agent designs with respect to point-of-reference artefacts is through automatically propagating these artefacts to lower-level designs without involving designers in the process. In the context of AOSE, there has been little research into propagating top-level information down into the low-level design artefacts. Although most AOSE methodologies offer limited information-propagation features across their phases as a consequence of their tool support, they do not support the sufficient propagation for a complete design. For example, Prometheus propagates the actions and percepts identified in the analysis overview into the detailed designs of the agents. However, it does not propagate full scenarios with the plans to realise them.

The Tropos methodology has many tools to support the methodology process and help with generating methodology artefacts [Morandini et al. 2007]. One of these tools is Tropos agent-oriented modelling (TAOM4e), which enables the designer to model the intended agent-based system. This tool has a code-generation feature that takes a detailed design and provides a skeleton code in the JADE programming language. The tool proposes a number of generators that facilitate the production of a skeleton code, which targets the JADE and Jadex agent platforms based on the detailed design artefacts or the goal model. The UML2JADE generator is used to generate a JADE agent code with respect to the agent interaction diagrams. The JADE agent code is generated by the transformation of the interaction diagrams' meta-model to the JADE meta-model, leading to the creation of an XML metadata interchange (XMI) file that helps to produce the capability files [Penserini et al. 2006]. The agent-interaction diagrams are propagated directly as a skeleton implementation code without considering the later phases of design that preclude the designer from being able to modify and control the protocol elements at a design level; for example, a plan that sends a particular message in a protocol may perform other tasks that need to be modelled at the design level.

In our previous work in [Abushark and Thangarajah 2013], we propose an approach for automatically propagating interaction protocols into agent-behaviour models. We present a mechanism, supported by algorithms, to automate and support the generation of detailed design artefacts with respect to the developed set of interaction protocols. Even though our preliminary evaluation showed that automatic propagation produces an error-free design, this approach seems infeasible, since more than 21 unique algorithmic cases were developed only for one construct (alternative). In addition, this approach does not consider scalability issues in cases where the intended design has multiple protocols. Furthermore, automation may affect the maintainability of the design, as it is normal for designers to alter their initial designs.

There have been proposals for automating the transformation between models in agent-oriented modelling [Bresciani et al. 2001, Perini and Susi 2005, García-Magariño et al. 2009]. These proposals support methodologies that adopt model-driven architecture (MDA) concepts and standards [Mellor 2004]. They require meta-models for both the target and source models, as it is essential in MDA to define the meta-models of source and target modelling notations based on a standard and also to state the transformation mechanisms between meta-model parts. In [Perini and Susi 2005], the authors propose a mechanism to automatically transform a Tropos plan decomposition model into a UML 2.0 activity diagram. Although such a transformation can ensure that the activity diagram in the detailed design phase is correct with respect to the plan decomposition model, it does not guarantee that the plan decomposition model is correct in the first place. Moreover—and similar to the propagation approach in Prometheus—such transformations may affect the maintainability of the design since software designers often use the model provided as a starting point, rather than as a final artefact.

In summary, existing approaches for verifying agent-based systems address different artefacts of the system of interest and apply verification automation at various levels. None of these approaches enforces a complete, nor automated, mechanism for verifying the internals of agents (i.e., agent-behaviour models) with respect to the point-of-reference artefacts (i.e., requirements and interaction protocols) at the design stage and without requiring source code. The approach of this thesis overcomes the limitations mentioned above.

CHAPTER 3

Conceptual Framework

This thesis proposes a verification framework that enables the early detection of defects in agent designs. This chapter describes the concepts and methods of our verification framework. We begin by explaining the design units considered in the framework (see Section 3.1). We ground our approach in Prometheus and we outline the design components of the Prometheus-based agents that are considered in our framework. Section 3.2 presents the overall architecture of the framework developed in this thesis. Section 3.3 highlights the types of defects that our approach can detect in agent designs. In Section 3.4, we discuss the concept of *design coverage*. In Section 3.5, we provide an overview of the technical method developed in this thesis.

3.1 Agent Design Units

The purpose of the *verification framework* is to *verify* agent designs against two point-of-reference artefacts: (1) scenarios and (2) interaction protocols. In Prometheus, agents are modelled such that they fulfil their roles in the scenarios and in terms of the protocols specified early in the development life-cycle. As mentioned in Section 2.4, scenarios in Prometheus represent basic runs in the desired system; each scenario includes a sequence of steps; these steps can be of different types (*actions, percepts* and *goals*). Interaction protocols show the legal interactions between many agents exchanging events (*percepts, actions* and *messages*).

Traditional software systems based on the common object-oriented paradigm have classes and objects as their core units. However, BDI-agent systems are designed in terms of agents, with

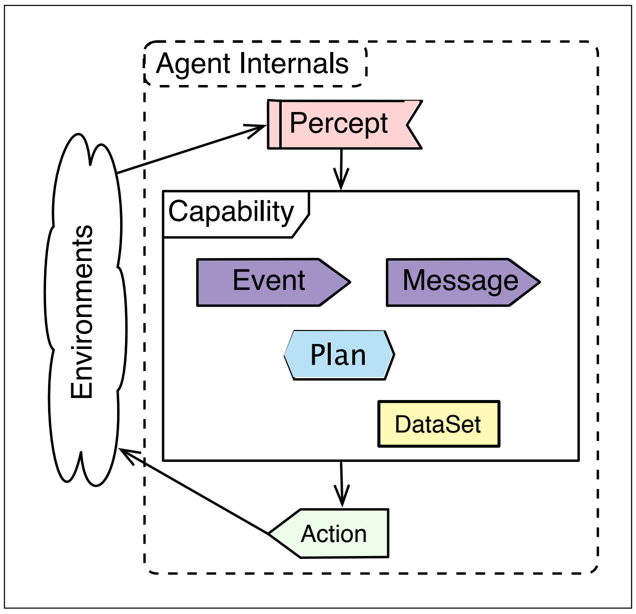


Figure 3.1: Design components of a Prometheus-based agent

each one having its own goals to achieve. In Prometheus, agents are modelled using *plans, events* and *datasets*. Goal realisation is achieved through *plans* that are triggered by events. Additionally, agents, when necessary, can store their beliefs in datasets. Figure 3.1 illustrates the design units of an agent that has been developed following the Prometheus methodology. These units are common to most agent designs that follow the BDI model of agency mentioned in Section 2.2.

This section discusses the three design units mentioned above (plans, events and datasets) and their verifiable features. Note that our verification framework considers the design-level view of the system-to-be—that is, it does not require implementation. Therefore, we make assumptions about the interpretation of some design concepts (e.g., choice in agent designs) because the framework lacks essential details (i.e., the implementation-level view) required for accurate interpretation.

Events

There are four types of events that a Prometheus-based agent can capture; namely, *percepts*, *actions*, *internal events* and *messages*. Percepts are external events that are received by an agent from the environment and need to be reacted to. Actions are the events to be posted by an agent to the environment that may affect that environment. Internal events are posted by the agent to itself

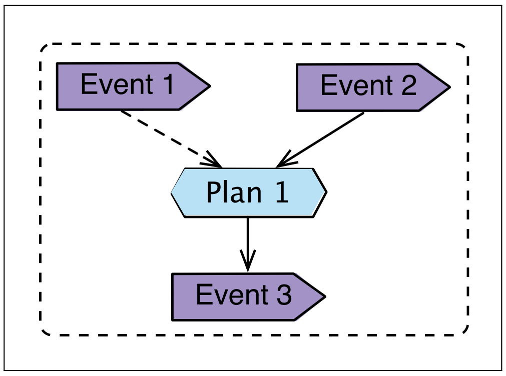


Figure 3.2: A plan with one event to process

to trigger sub-tasks or realise other goals. Message events are used to exchange information between different agents. These event types, excluding the internal events, have a direct link to the point-of-reference artefacts considered in the proposed approach (i.e., scenarios and interaction protocols). In fact, percepts, actions and goals represent the basic units in scenarios. In addition to percepts and actions, protocols can also capture messages through their structure. Even though internal events are not directly linked to the point-of-reference artefacts, they may represent the realisation of goal steps in scenarios or ensure the flow of communications between agents. Therefore, all four types of event are included as basic design units in our approach.

Plans

Plans are step-by-step *recipes* that allow agents to achieve their goals (see Section 2.2). A plan must have only one triggering event that initiates its execution. However, Prometheus allows plans to handle multiple events, with the triggering event to be processed first. In some cases, plans need to wait for inputs other than their triggers (e.g., messages) that help such plans to achieve their assigned goals. Plans may also post multiple events: internal events to trigger other tasks, messages to capture communication and actions to affect the environment.

Figure 3.2 shows that 'Plan 1' is triggered by 'Event 1' (shown by a dashed arrow from the event to the plan), processes 'Event 2' (shown by a solid arrow from the event to the plan) and posts 'Event 3' (shown by a solid arrow from the plan to the event). As per the figure, 'Plan 1', at some point in its execution, waits for 'Event 2' to be processed; however, it is certain that 'Event 1' will be handled before 'Event 2' is processed. The plan also posts 'Event3', but the order between handling 'Event 2' and posting 'Event 3' cannot be determined.

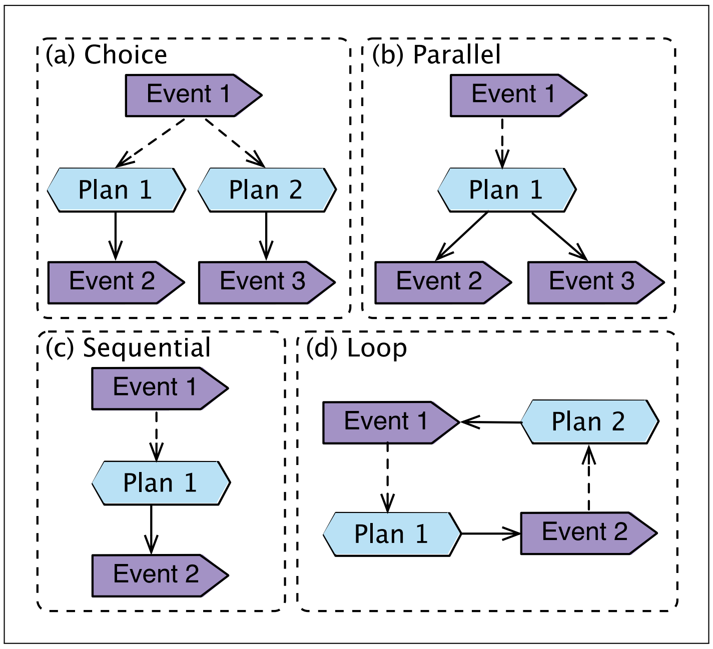


Figure 3.3: Event/plan relationships

Events can be associated with plans forming four different control fragments: *sequential, choice and parallel* and *loop* (see Figure 3.3). The sequential control fragment is formed when an event is handled by a single plan that posts only one event (see Figure 3.3 (c)).

An event can be a trigger for multiple plans where only one plan is to be executed, based on the context conditions of the plans, forming the choice control fragment. As shown in Figure 3.3 (a), ‘Event 1’ is handled by two plans (‘Plan 1’ and ‘Plan 2’), but only one plan is to be executed.

Assumption 1 (choice in agent designs)**.** If an event is handled by multiple plans, we assume *choice* between these plans.

A plan can post multiple events and the framework of this thesis assumes that such events are to be posted in parallel. ‘Plan 1’ in Figure 3.3 (b) posts ‘Event 2’ and ‘Event 3’ in unspecified order. Thus, ‘Event 1’ can be posted before ‘Event 2’, or ‘Event 2’ before ‘Event 1’. Although at the implementation level and through selection statements (e.g., *if/then/else*) a programmer can implement a choice between these events, we permit any behaviour consistent with interleaving parallelism in this case, since we are considering a static view of the system-to-be.

Assumption 2 (parallelism in agent designs)**.** If a plan posts multiple events, we assume a parallel relationship between these events.

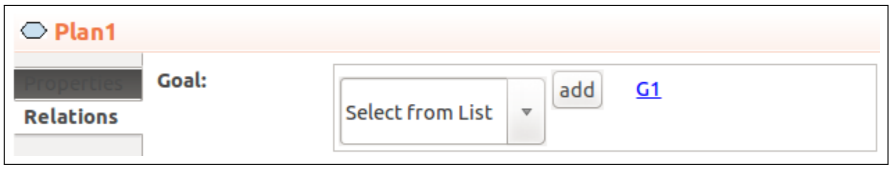


Figure 3.4: Plan descriptor

The loop-control fragment captures the interactions between plans where a plan posts an event that is handled by one of its ancestor plans, as illustrated in Figure 3.3 (d). Because of the lack of implementation-level details, we cannot identify the number of iterations a loop should make. Thus, we consider only one iteration of any loop-control fragment.

Assumption 3 (loops in agent designs)**.** If a plan posts an event that is handled by one of its ancestor plans, then a loop fragment is formed (i.e., a *cycle* between the plan and its ancestor). As a consequence of the lack of the implementation-level details, loops are assumed to iterate only once and, hence, all cycles are eliminated from agent-behaviour models. This assumption approximates the number of times a loop-control fragment can iterate. We exclude cases when the loop iterates *zero* times, as here the design under investigation captures the fragment and this implies that the loop is implemented in the design. Given this, our approximation is sound but incomplete.

Each plan in Prometheus has a descriptor that allows designers to specify the *goal/s* (i.e., the steps in the scenarios) to be realised. Figure 3.4 depicts part of the descriptor of 'Plan 1' in Figure 3.2. As can be seen from Figure 3.4, 'Plan 1' is responsible for realising goal 'G1'. As the framework considers the design-level view of the system-to-be, it cannot be determined whether the intended plan was successfully executed. In other words, we cannot determine whether the goals assigned to the plan have been achieved. Thus, we do not assume that plans achieve their assigned goals. Instead we assume that goals are realised by plans.

Assumption 4 (goals at design level)**.** Since we do not have access to the implementation of the system-to-be, we cannot judge its execution. Hence, we do not assume the achievement of goals in our approach. Instead, we assume that goals are handled by plans at a design level. This assumption is consistent with the description of goals in the Prometheus methodology [Padgham and Winikoff 2005].

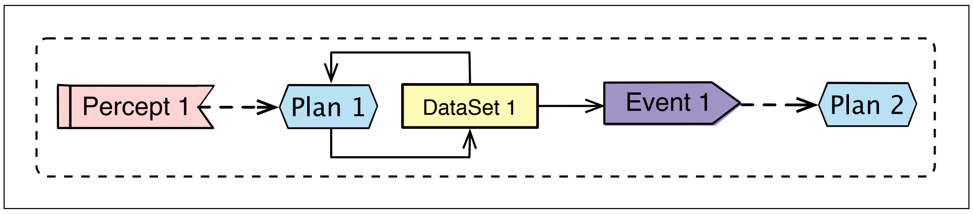


Figure 3.5: Datasets processing example

Datasets

Datasets are design units that represent the agent's beliefs (see Section 2.2). Agents manipulate their beliefs by having plans that insert, update and delete data. As shown in Figure 3.5, agents can write to datasets (shown by a solid arrow from the plan to the dataset) and read from datasets (shown by a solid arrow from the dataset to the plan).

In Prometheus, a *dataset* can perform actions based on certain changes in its status. For example, the insertion of a new record into an agent's belief may automatically post events that activate other plans. In Figure 3.5, 'Data-Set 1' may post the internal event 'Event 1', which triggers 'Plan 2'. In the approach proposed by this thesis, datasets are considered only when they post events. This is because such datasets directly contribute to the behaviour of agents, whereas datasets that do not post events are concerned with writing and reading data—procedures that our approach does not check.

3.2 The Conceptual Model

The verification framework proposed by this thesis takes into account two main design artefacts: point-of-reference artefacts and low-level agent-behaviour models. In the context of Prometheus, the point-of-reference artefacts considered in our approach are *scenarios* and *protocols*.

Since each step in a scenario is associated with a *role* to realise that step, the *role-grouping model* of the system-to-be is considered part of the point-of-reference artefact. Further, the *goal-overview diagram* of the system-to-be is taken into account as an input, as scenarios may include goal steps. Given this, the framework takes scenarios as primary artefacts, whereas it takes the *goal-overview diagram* and the *role-grouping model* of the system-to-be as complementary artefacts to scenarios. The low-level agent-behaviour models considered in our approach are repre-

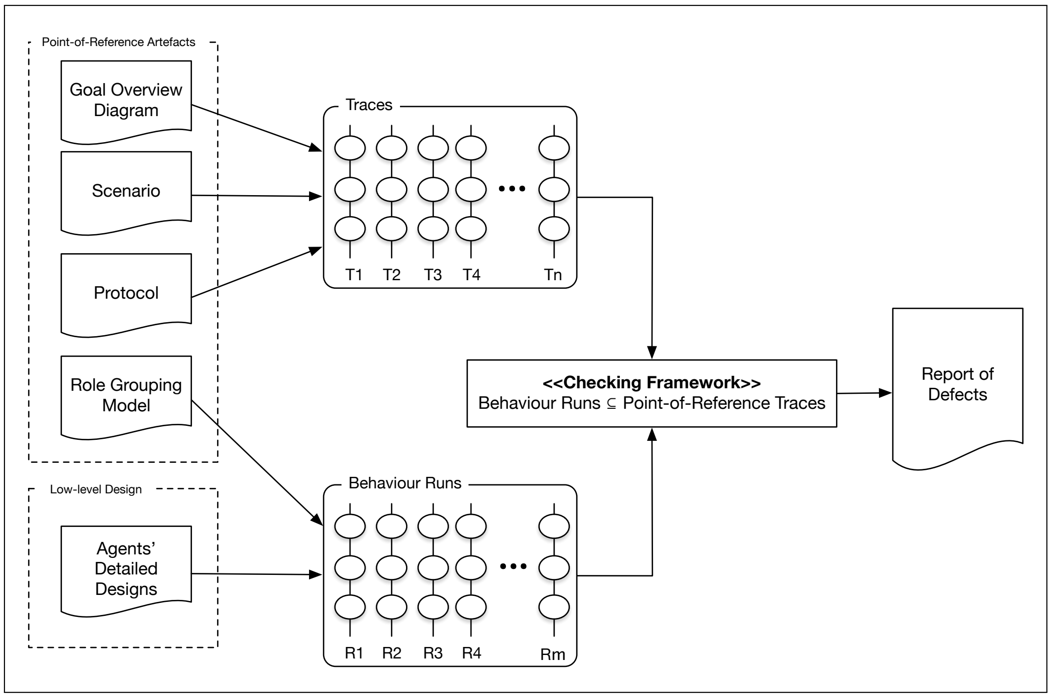


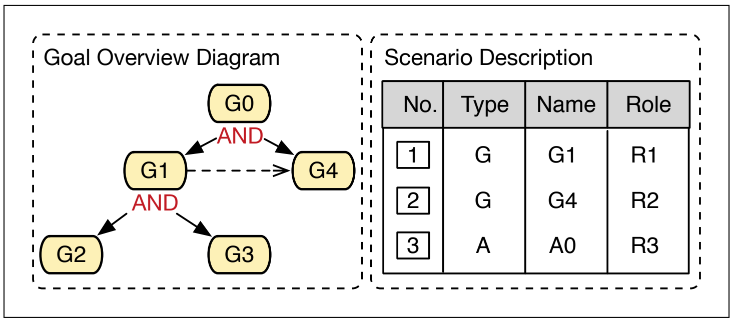
Figure 3.6: Static verification conceptual framework

sented by the *agent detailed design overview diagrams*.

The framework uses these artefacts to produce a set of traces from both the point-of-reference artefact and the agent-behaviour models to be automatically checked and verified. The output of this verification framework is a report that logs the potential defects in the agent-behaviour models with respect to the specified point-of-reference artefact.

Figure 3.6 depicts a general overview of the proposed verification framework. The framework accepts one point-of-reference artefact at a time (i.e., one scenario or one protocol at a time) as an input, allowing software engineers to focus on performing checks for the artefact under development. In fact, the Prometheus methodology provides systemic and structured support for the *iterative/incremental* development approach [Perepletchikov and Padgham 2005]. Note that, since it is possible for the designer to test the design at any incremental stage, the less complete the agent design, the more defects may be identified.

As Figure 3.6 shows, any Prometheus-based design has four core elements: scenarios, a goal model, protocols and agent-behaviour models. In this section, we define these elements. Further, we explain what it means for agent-behaviour models to conform to the specified point-of-reference artefact.



No.	Type	Name	Role
1	G	G1	R1
2	G	G4	R2
3	A	A0	R3

Figure 3.7: Goal step realisation example

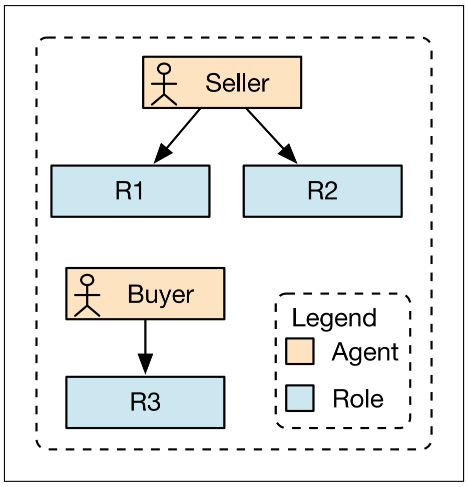


Figure 3.8: Part of the role grouping model relevant to the scenario in Figure 3.7

3.2.1 Point-of-Reference Artefacts

Our verification framework considers two point-of-reference artefacts—scenarios and protocols—to which the agent-behaviour models must conform. Thus, we assume the correctness and completeness of these point-of-reference artefacts.

Assumption 5 (point-of-reference artefacts)**.** We assume that the point-of-reference artefacts—the scenarios, the protocols and the goal model—are correct and complete.

Scenarios

A scenario is a sequence of steps that the agents in the system-to-be need to realise (see Figure 3.7). These steps can be one of the following types: *percept*, *action* or *goal*. Each step in a scenario must be associated with an agent *role*, which is the entity responsible for the step. These roles are then linked to the agents in the system-to-be (see Figure 3.8).

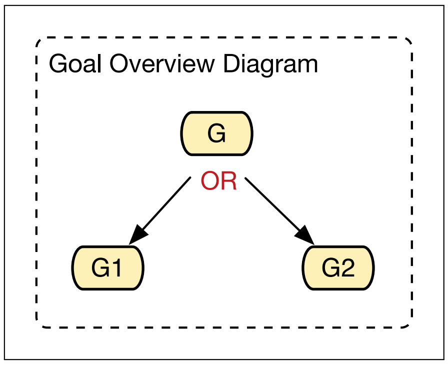


Figure 3.9: OR-decomposition goal overview diagram

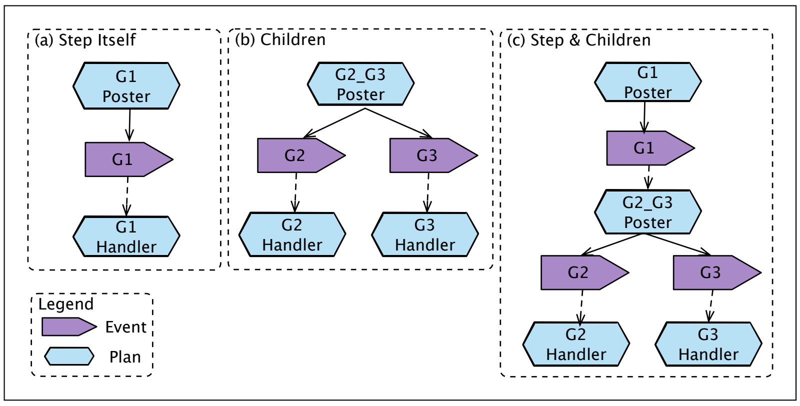


Figure 3.10: Different ways to realise a goal step

Although a scenario is a single sequence of steps, there may be different ways to realise the same scenario. This is because, when there are goal steps, the goals may also be realised through their sub-goals from the goal-overview diagram.

The goal-overview diagram is a *goal tree*, whose nodes are the goals that need to be pursued. The branches in the goal-overview diagram show the relationship between goals, including how goals are decomposed into sub-goals. As discussed in Section 2.4 of Chapter 2, there are three decomposition types. The first is the *'OR'* decomposition, which implies that a parent goal is realised if one of its children is realised.

Assumption 6 (*OR* goal-decomposition)**.** We assume that the *OR*-decomposition is an *exclusive OR*. This means that one—and only one—child of an OR-decomposed goal step needs to be implemented to realise the step. For example, the design that implements both children 'G1' and 'G2' in Figure 3.9 is considered a defective design because it achieves the goal 'G' twice.

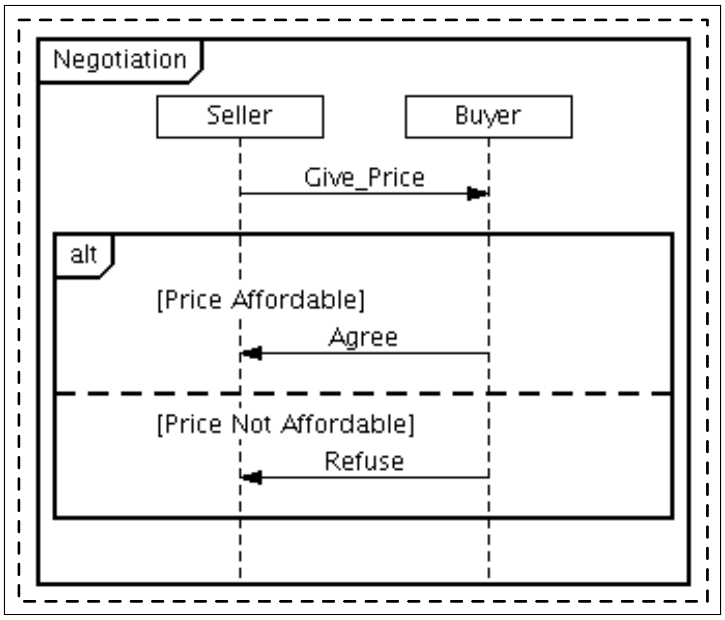


Figure 3.11: Simple interaction protocol

The second from of decomposition is the *'directed-AND'* (meaning that a parent goal is realised if all its children are realised in a specified order). The third form is the *'AND'* decomposition (meaning that a parent goal is achieved if all its children are realised in any order). For example, the goal step 'G1' of the scenario in Figure 3.7 can be realised through the step in the scenario itself (see Figure 3.10 (a)). Alternatively, it can be realised by the step with its children in any order (see Figure 3.10 (b)), or just by the children, ('G2' and 'G3') in an unspecified order (see Figure 3.10(c)).

Since we need to consider relevant information from the goal tree to a goal step within a scenario S, this scenario may result in a set of requirement traces, denoted by $[[T_S]]$. Specifically, $[[T_S]]$ is a set of sequences that includes the original sequence of steps comprising the scenario and its alternatives, as per Example 1.

Example 1. The scenario in Figure 3.7 yields:
$[[T_S]] = \{ \langle G0, A0 \rangle, \langle G0, G1, G4, A0 \rangle, \langle G0, G1, G2, G3, G4, A0 \rangle, \langle G0, G1, G3, G2, G4, A0 \rangle,$
$\langle G1, G4, A0 \rangle, \langle G1, G2, G3, G4, A0 \rangle, \langle G1, G3, G2, G4, A0 \rangle, \langle G2, G3, G4, A0 \rangle,$
$\langle G3, G2, G4, A0 \rangle \}$

A design that captures any sequence in $[[T_S]]$ would realise the scenario in Figure 3.7. For example, a design that implements the original sequence $\langle G1, G4, A0 \rangle$, which belongs to $[[T_S]]$, realises the scenario. Further, a design that captures the sequence $\langle G2, G3, G4, A0 \rangle$ also achieves the scenario, as the goal step $G1$ in the scenario can be realised through its children (G2 and G3) from the goal tree.

Protocols

Our approach is grounded in the Prometheus methodology; therefore, it assumes that protocols are specified using AUML-sequence diagrams (see Section 2.4). AUML-sequence diagrams specify the allowable *sequences of messages* between agents. Protocols may capture many execution flows through capturing different constructs (see to Section 2.3.2).

For example, the interaction protocol in Figure 3.11 results in three possible flows because it has an alternative construct with two executable regions.

Example 2. Given the above, let $[[T_{IP}]]$ be the set of sequences resulting from the protocol in Figure 3.11. Then:
$[[T_{IP}]] = \{ \langle Give\_Price \rangle, \langle Give\_Price ,Agree \rangle, \langle Give\_Price ,Refuse \rangle \}$

Unlike the goal steps in a scenario, each sequence of $[[T_{IP}]]$ in Example 2 is an execution flow, whereas the traces of $[[T_S]]$ in Example 1 are alternative ways to accomplish a single execution flow in a scenario. In other words, a design that fully covers the protocol in Figure 3.11 ***must*** implement all the three sequences in $[[T_{IP}]]$. However, a design that implements more than one sequence from $[[T_S]]$ is considered a ***repetitive*** design (i.e., a defective design). Note that we do not assume that designs need to fully adhere interaction protocols; however, we do offer a method for checking their compliance.

3.2.2 Agent-Behaviour Models

To verify an agent design against a particular point-of-reference artefact, we compare all possible behaviour runs that may be exhibited by the agents in the system-to-be at run time with the specified point-of-reference artefact.

The agent-behaviour models comprise entities that are captured by both scenarios and protocols (i.e., percepts, actions, goals and messages) in addition to plans. Plans are the executable units

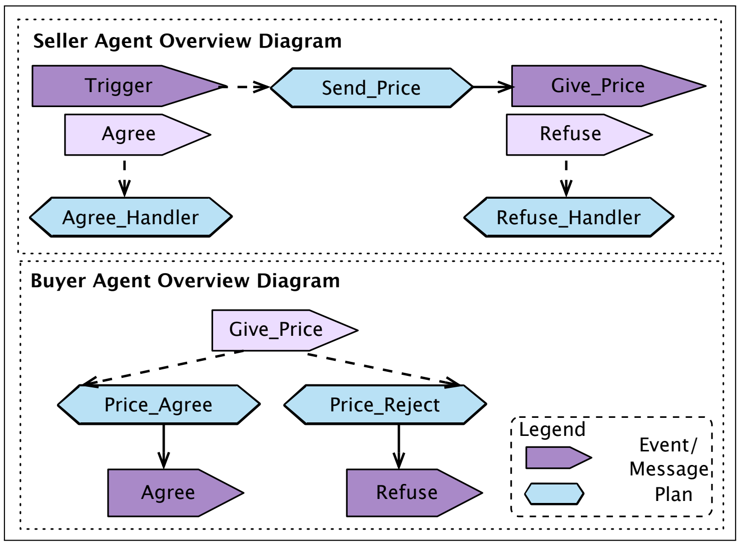


Figure 3.12: Agent-behaviour models of the protocol in Figure 3.11

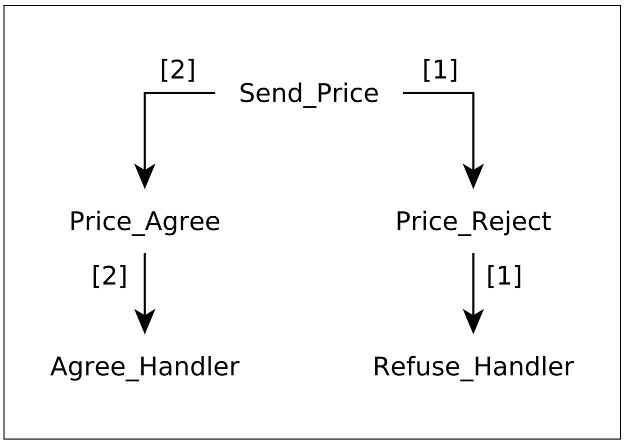


Figure 3.13: Graph of the plans in Figure 3.12 (note that the graph has two paths)

in the agent-behaviour models; hence, they generate the agents' behaviour. Thus, a behaviour run is a sequence of plans.

In the context of agent design, a plan is defined in terms of its trigger, a set of inputs and a set of outputs. The trigger for a plan and its inputs can be any design entity except actions, while its output set can contain any design entity aside from percepts. Thus, a *plan* is a tuple where $p = \{N, I, O, T_r\}$, in which N is a unique name that identifies the plan within the design; I is a set of input events for the plan to be processed and must not include actions; O is a set of output events that a plan can post, which must not include percepts; T_r is the *trigger* of the plan where $T_r \in I$ and each plan must have a trigger (i.e. $\forall\, p\, \exists! T_r \mid T_r \notin \mathit{actions}$).

Plans are linked with each other through their outputs, forming a *graph* of plans. For example, the ‘Send_Price’ plan shown in Figure 3.12 is initiated by the ‘Trigger’ event and posts the ‘Give_Price’ message (i.e., its output set has only one element). The ‘Give_Price’ message is the trigger for two plans: the ‘Price_Agree’ and the ‘Price_Reject’ plan. The ‘Price_Agree’ plan sends the ‘Agree’ message to be handled by the ‘Agree_Handler’ plan, whereas the ‘Price_Reject’ plan sends the ‘Refuse’ message to be handled by the ‘Refuse_Handler’ plan. As can be seen from the example, all five plans in Figure 3.12 are linked by their outputs. Figure 3.13 shows the *graph* based on the associations between the five plans in Figure 3.12. Recalling the different relationships between events and plans as discussed in Section 3.1, graphs of plans can capture four control fragments: *sequential* (SEQ), *choice* (ALT), *parallel* (PAR) and *loops*:

1. Sequential (SEQ): $SEQ(p_1,\ldots,p_n)$ denotes that the plans within this fragment are to be executed sequentially.

2. Alternative (ALT): $ALT(p_1,\ldots,p_n)$ denotes that only one of the plans within this fragment is to be executed.

3. Parallel (PAR): $PAR(p_1,\ldots,p_n)$ denotes that the plans within this fragment are to be executed in parallel. In other words, this control fragment indicates that all the plans are active at the same time, and hence, their execution will be interleaved.

4. Loop (cycles): Cycles are specified by the posting of another event by the plan that handles an initial event and that will also handle the subsequent, new event that it has just posted ($p_1 \to e_1 \to p_1$). However, as mentioned earlier in Section 3.1, we treat cycles as having one iteration.

Given the above, if PG denotes the graph in Figure 3.13, then this graph is textually described as follows:

$PG = SEQ(Send\_Price, ALT(SEQ(Price\_Agree, Agree\_Handler), SEQ(Price\_Reject, Refuse\_Handler)))$

While a plan is being executed, it processes its input design entities and produces its output design entities. Thus, we can obtain possible behaviour runs in terms of sequences of design entities (i.e., events, messages, percepts, actions and goals) by substituting plans with the sets of events for their inputs and outputs. For example, by substituting the plans in *PG* above with their input and output sets, we obtain the following set of traces (i.e., behaviour runs without plans) denoted by $[[B_r]]$:

Example 3. $[[B_r]] = \{ \langle\ Give\_Price\ , Agree\ \rangle, \langle\ Give\_Price\ , Refuse\ \rangle\ \}$

Since plans in an agent design may form the aforementioned four control fragments, it is intuitive that a given design may capture many behaviour runs. For example, the graph in Figure 3.13 captures two possible execution runs, as it illustrates an alternative control fragment.

3.2.3 Checking the Conformance of Agent Designs

Agent designs are graphs that show the relationships between plans. In our approach, we check the correctness of these designs, ensuring that the set of behaviour runs (e.g., $[[B_r]]$ in Example 3) form a subset of the set of the traces (e.g., $[[T_{IP}]]$ in Example 2). Conceptually, a behaviour run is said to conform to a trace if the run contains the *exact* elements of the trace in the same *order*.

Property 1 (correctness)**.** An agent design is said to be correct concerning the specified point-of-reference artefact *iff* $[[B_r]] \subset [[T_S]]$, where $[[B_r]]$ is the set of behaviour runs from the agent-behaviour models and $[[T_S]]$ is the set of traces of the specified point-of-reference artefact.

Considering the first run of $[[B_r]]$ in Example 3, which has two tokens: 'Give_Price' and 'Agree', the run matches the second trace, or sequence, of $[[T_{IP}]]$ in Example 2. Thus, we can say that the run is valid concerning the trace. Similarly, the second run in $[[B_r]]$ conforms to the third trace of $[[T_{IP}]]$. However, if the first token in the second run ($\langle\ Give\_Price\ , Refuse\ \rangle$) is 'Refuse' instead of 'Give_Price', then the run will not conform to any trace of $[[T_{IP}]]$, as the tokens are not in the same order. Since all the runs in $[[B_r]]$ appear in $[[T_{IP}]]$ (i.e., $[[B_r]] \subset [[T_{IP}]]$), the designs in Figure 3.12 conform to the protocol in Figure 3.11.

	Failure (matching with the point-of-reference traces)	**Cause (in the design)**
1	The behaviour run is completed, while the trace involved in the comparison is not entirely completed (i.e. traces have more tokens to compare with).	1. The run contains fewer *design entities* (i.e. steps or messages) than it should, relative to the point-of-reference artefact.
2	The behaviour run is not fully completed, while the trace involved in the comparison is completed.	2. The behaviour run contains more *design entities* (i.e. steps or messages) than it should, relative to the point-of-reference artefact.
3	The behaviour run is relevant to the traces, but it did not match any trace.	3.(a) The *design entity* (i.e. a step or a message) in the run that needs to be matched with the current token in the trace is missing; or 3.(b) The ordering between *design entities* (i.e. steps or messages) within the run is inconsistent with the point-of-reference artefact.

Table 3.1: Categorisation of causes for failures

3.3 Types of Defects in Agent Designs

Our approach marks a run as *'passed'* if and only if the tokens in the run are consistent with the tokens for a trace from the point-of-reference traces. Otherwise, the run is recorded as *'failed'*. To provide informative feedback about the agent-behaviour models, we identify a cause for a failure. Table 3.1 lists all possible failures and their causes. We use the scenario in Figure 3.7 (see $[[T_S]]$ in Example 1) and its possible designs as shown in Figure3.14 to explain these failures.

The causes in Table 3.1 indicate the possible types of defect that our framework can detect in an agent design. We now consider the three cases of failure:

Case 1: run misses design entities — The first failure occurs when the entire run is consumed (i.e., all the tokens in the run are used in the comparison), while no match with any trace from the point-of-reference traces is reported. This failure indicates that the design **misses** the entity that is supposed to be realised according to the specified point-of-reference artefact. For example, the design in Figure 3.14 (b) results in a single run: $\langle G1, G4 \rangle$. However, all sequences in the set of traces $[[T_S]]$ have the action step 'A0' as a token. Thus, for the run to pass the comparison it has to have that action step ('A0'). This failure, based on the pre-

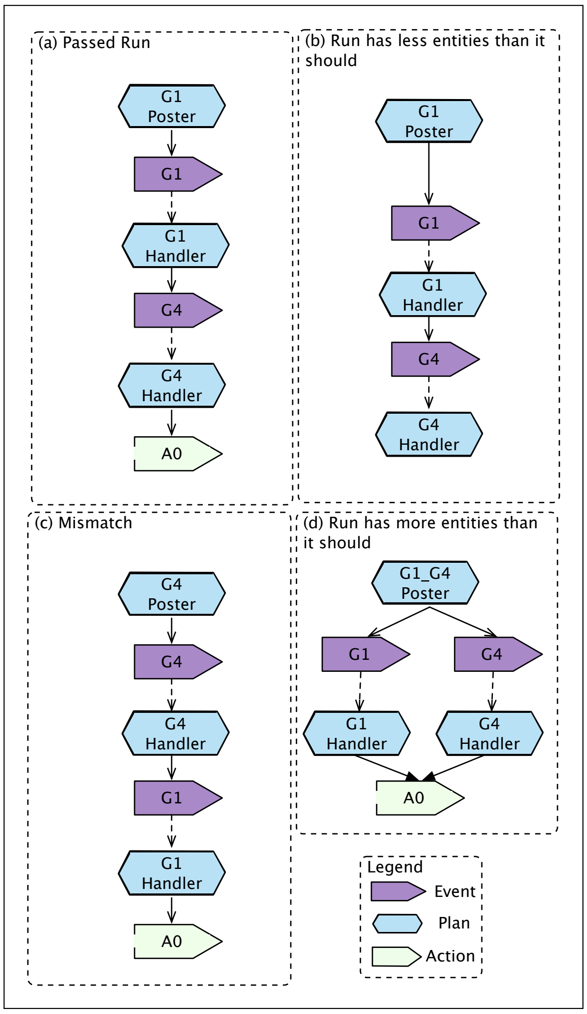


Figure 3.14: Examples on defects in agent-behaviour models

vious example, highlights that the design element that is relevant to the point-of-reference under investigation **misses** the action step ‘A0’ in the scenario.

Case 2: run has more design entities than it should — In this case, the failure is because the run has more tokens than it should. This means a design entity appears in the same run multiple times, whereas the specified point-of-reference artefact requires such entities to appear only once. For example, the plans ‘G1 Handler’ and ‘G4 Handler’ in the design shown in Figure 3.14 (d) post the action ‘A0’. Since both plans interleave (i.e., both are executed), the action ‘A0’ is posted twice. Hence, all runs capture ‘A0’ twice (e.g., $\langle\, G1, G4,\ A0,\ A0\,\rangle$), while none of the traces in $[[T_S]]$ has ‘A0’ twice. Comparing the run with its relevant traces, the run matches the trace until it consumes the first instance of ‘A0’. Then, the trace is also entirely consumed, while the run still has another instance of ‘A0’ to consume.

Case 3: mismatch between the run and its relevant traces — The failure, in this case, occurs when the comparison operation between a run and its relevant traces is obstructed because of a mismatch between the tokens of both the run and the traces. This mismatch indicates either that the entity is **missing** from the design or that the **ordering** between design entities in the run is erroneous with respect to the point-of-reference artefact. For example, the design in Figure 3.14 (c) results in the following run: $\langle\, G4,\ G1,\ A0 \,\rangle$. Even though the run includes the correct design entities with respect to the fifth sequence in $[[T_S]]$ ($\langle\, G1,\ G4,\ A0 \,\rangle$), it does not match the ***occurrence order*** of the tokens captured by the trace; hence, it is marked as a failed run.

The missing design entities, in this case, are different from those in Case 1. In Case 1, the run is consumed, while the trace still has further tokens to be compared. However, in this case, the run is not fully consumed; however, the comparison process is obstructed because the current token of the run does not match the token of the trace. Unlike in the case of miss-ordering, the token is not in the run. For example, let us assume that the run extracted from the design in Figure 3.14 (a) is $\langle\, G1,\ A0 \,\rangle$. According to the first token in the run, this run should be compared with the fifth trace in $[[T_S]]$ ($\langle\, G1,\ G4,\ A0 \,\rangle$). The first token in the run (‘G1’) matches the first token in the trace. However, the second token in the run is ‘A0’, which does not match the second token in the trace (‘G4’). As a result, the run is marked as failed. As per the example, the comparison process is obstructed because of a mismatch between the particular token of the run and the token of the trace. However, the run ***misses*** ‘G4’.

3.4 Checking for Design Coverage

Our framework does not only provide the ability to detect potential defects in a design concerning the specified point-of-reference artefact, it also *checks* whether the design fully covers that artefact. A design that faithfully conforms to its point-of-reference artefact is not necessarily complete with respect to that artefact.

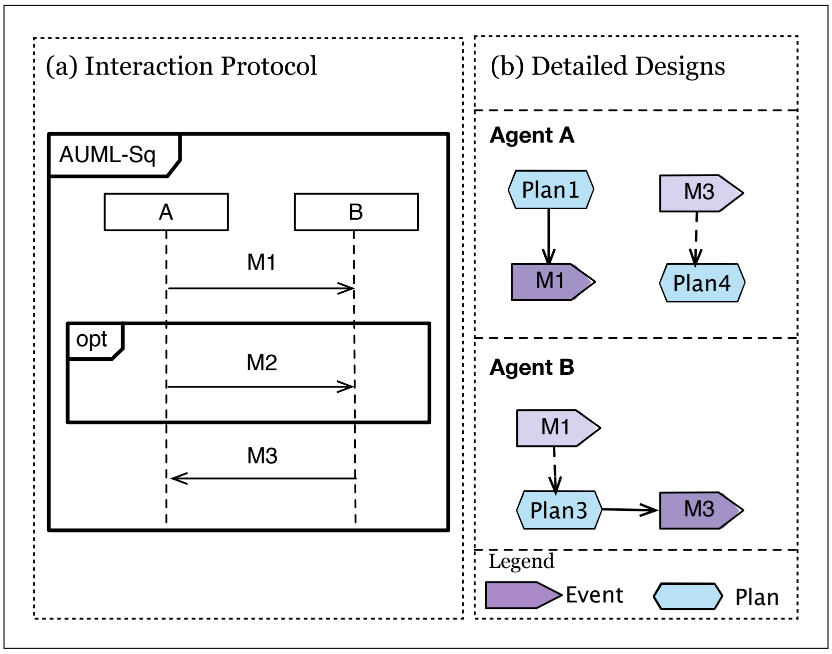


Figure 3.15: An interaction protocol and its equivalent design

Property 2 (completeness)**.** A point-of-reference artefact is said to be fully covered by the agent-behaviour models under investigation *iff* $[[B_r]] \subseteq [[T_S]]$, where $[[B_r]]$ is the set of behaviour runs of the agent-behaviour models and $[[T_S]]$ signifies the set of traces of the specified point-of-reference artefact.

As stated earlier in Section 3.2.1, a point-of-reference artefact results in a set of traces. Such a set contains sequences of entities (steps in a scenario or communication constructs). These sequences are usually *alternative ways* to realise the same execution. However, in some cases, these sequences represent *execution flows* and, hence, the design needs to ***cover*** them all to be a ***complete*** design. For example, the protocol in Figure 3.15 (a) captures two execution flows as follows: $\langle M1, M2, M3 \rangle$ and $\langle M1, M3 \rangle$. The design of this protocol (in Figure 3.15 (b)) captures only one behaviour run: $\langle M1, M3 \rangle$. By comparing this run with the two traces of the protocol, the outcomes will produce a 100% passing rate, as the run conforms to the second trace. Although it is clear from the designs shown in Figure 3.15 (b) that 'M2' is missing, no defects will be flagged in the design, since the set of runs for the design is a *subset* of the set of traces.

Similarly, when the point-of-reference artefact is a scenario, a design may fully conform to it, but may miss a variation—for example—an execution flow in that scenario.

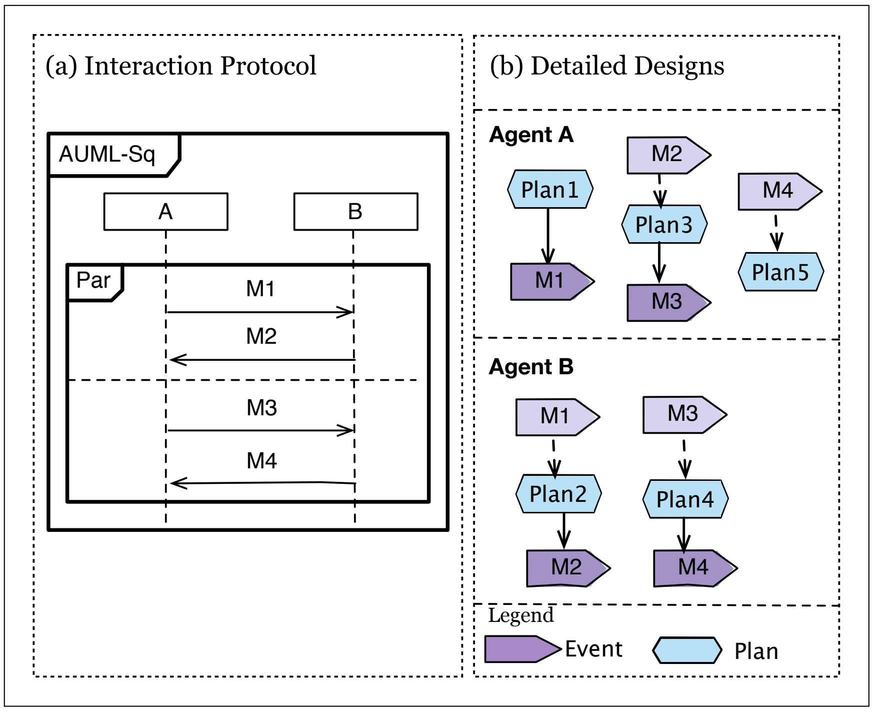


Figure 3.16: A protocol with PAR construct and its equivalent design

The framework does not flag the lack of coverage, if it exists, as a defect. Instead, it reports it as a warning that may require the designer's attention, as the designer may choose not to implement the entire point-of-reference artefact. This coverage check ensures the completeness of the relevant parts of the design to the point-of-reference. In other words, it verifies that all execution flows—and not the possible alternative flows—of the specified point-of-reference artefact are covered by the design. For example, the protocol in Figure 3.16 (a) results in six sequences as a result of the interleaving between the two regions of the parallel construct (we denote the set of traces by $[[T_{PAR}]]$):

$[[T_{PAR}]] = \{\langle M1, M2, M3, M4\rangle, \langle M1, M3, M2, M4\rangle, \langle M1, M3, M4, M2\rangle, \langle M3, M4, M1, M2\rangle,$
$\langle M3, M1, M2, M4\rangle, \langle M3, M1, M4, M2\rangle\}$

The sequences in $[[T_{PAR}]]$ form possible alternative ways to execute the parallel construct in the protocol. Thus, a design with all its runs matching these six sequences in $[[T_{PAR}]]$ (i.e., the set of runs is a subset of $[[T_{PAR}]]$) is said to conform to the protocol. In other words, our approach considers only one form of interleaving and not all possible forms of interleaving in the cases of *parallel* control fragments. For example, the designs illustrated in Figure 3.16 (b) result in a single run and, hence, the set of runs (denoted by $[[B_r]]$) contains only one sequence: $\langle M1, M2, M3, M4\rangle$.

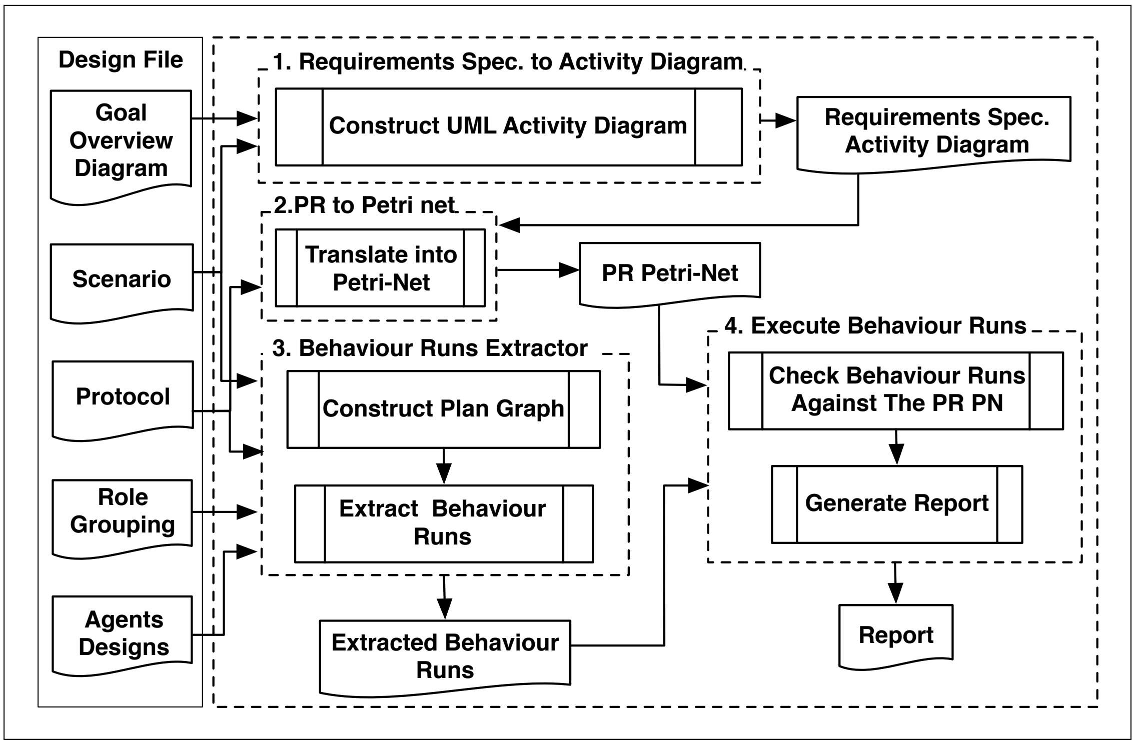


Figure 3.17: Static verification framework (PR: point-of-reference artefact, PN: Petri net)

Since the set of runs is actually a subset of the set of traces (see Property 1), then it can be said that the design in Figure 3.16 (b) conforms to the protocol in Figure 3.16 (a).

3.5 Technical Method Overview

This section summarises the technical aspects of our verification framework for implementing the conceptual framework described in this chapter. Figure 3.17 depicts the overall architecture of the proposed verification framework. As Figure 3.17 demonstrates, the framework encompasses four modules, each of which is responsible for generating a component according to the specified point-of-reference artefact as follows:

1. *Requirement specification to activity diagram transformer* (see Chapter 4). This module is responsible for automatically constructing an activity diagram from the specified scenario as well as the goal-overview diagram of the system-to-be.

2. *Point-of-reference to Petri net converter* (see Chapter 5). This module is responsible for translating the point-of-reference artefacts into Petri nets to act as executable models to

obtain the possible traces (i.e., $[[T_S]]$) .

3. *Behaviour runs extractor* (see Chapter 6). This module produces the set of all possible behaviour runs with respect to the specified point-of-reference artefact (i.e., $[[B_r]]$).

4. *Behaviour runs checker* (see Chapter 6). The previous modules result into two verifiable components, namely: the point-of-reference's Petri net and a set of behaviour runs. This module is responsible for checking each behaviour run against the point-of-reference Petri net by executing the run over the Petri net and reporting any discrepancies. In other words, this module checks the validity of Property 1 and also checks the completeness of the agent-behaviour models as concerns the specified point-of-reference artefact (i.e., Property 2).

Summary

This chapter introduced the concepts and methods of the verification framework proposed by this thesis. The framework checks the agent-behaviour models against two point-of-reference artefacts: scenarios and interaction protocols. The framework considers one point-of-reference artefact at a time along with the agent-behaviour models as inputs and generates a text-based report on the potential defects of the agent-behaviour models with respect to the specified point-of-reference artefact.

Conceptually, the framework checks that the set of behaviour runs emerging from the agent-behaviour models is a subset of the specified point-of-reference traces. The framework translates the point-of-reference artefact to a Petri net to act as an executable model. Then, it extracts all possible behaviour runs out of the agent-behaviour models and executes each behaviour run against the point-or-reference artefact Petri net, logging any discrepancies in the execution of the Petri net. A failure in executing a behaviour indicates an inconsistency between the design and the point-of-reference. Three types of defects are identified in our framework, which may affect agent-based designs as follows: missing entities, wrong ordering between entities and unnecessary repetitions of entities. Further, the framework ensures that the part of the design that is relevant to the specified point-of-reference artefact is complete.

CHAPTER 4

Requirements Specification via Activity Diagrams

The verification framework proposed in this thesis aims to check the conformance of the detailed plan structures of agents to two point-of-reference artefacts: scenarios and interaction protocols. In Prometheus (see Chapter 2, Section 2.4), the requirements of the system are specified via *scenarios* and *goals*. A scenario is similar to a use case in the object-oriented paradigm [Jacobson et al. 1992] and describes a particular run of the system as a sequence of steps. These step types include percepts[1], actions or goals. Goals can be decomposed into sub-goals, using a goal model. The combination of the scenarios together with the goal trees, forms part of the requirements for the system.

There are three limitations in the current representation of requirements in Prometheus:

1. Scenarios only capture a sequence of steps, which means that steps that should be performed in parallel cannot be specified.

2. Variations to the scenario are captured informally as English text.

3. The scattered nature of the requirements between scenarios and the goal model requires increased mental effort from designers to check for coverage of the requirements and can lead to potential design errors.

[1]Events that represent inputs into the system from the environment

To overcome these limitations, we propose the use of an *activity diagram* to complement the process of specifying requirements in Prometheus. It is important to note that our proposal can be generalised to any agent-oriented methodology that shares the same concepts as Prometheus.

This chapter walks the reader through our proposal of using UML-activity diagrams as more structured representations of requirement specifications to *complement* the process of the Prometheus methodology. Scenarios and goal-trees form an integral part of the Prometheus methodology. Hence, our proposal is not to replace these design artefacts, but to *complement* them with activity diagrams as a way of overcoming their current limitations. We do this by taking a scenario and its corresponding goal-trees[2] and generating an equivalent activity diagram (see Section 4.1). The activity diagram generated represents and models the flows of only one scenario at a time. Section 4.1.2 details our implementation of this generation process.

By using activity diagrams, we can ensure that the executable model of the requirements is complete, as they overcome the limitations Prometheus has when specifying requirements. Our proposal allows designers to illustrate which steps in the specified scenario can be attempted in *parallel* or to specify that the ordering does not matter. Further, activity diagrams permit designers to specify *variations* to a given scenario in a structured manner. Section 4.2 justifies our proposal and outlines the potential benefits of activity diagrams and how they overcome the limitations of the current approach in Prometheus.

We also performed a series of evaluations on 15 participants, who were tasked with interpreting and modifying two different sets of requirements one with an activity diagram and one without measuring their performance and asking for qualitative feedback (see Section 4.3). The reason for conducting this user study is to measure the impact an activity diagram has on the ability of a software engineer to understand and maintain requirement models. The results presented in Section 4.4 demonstrate that the participants were able to complete the tasks more correctly and more quickly using an activity diagram and that they unanimously preferred the addition of activity diagrams.

There have been a number of proposals to use activity diagrams when specifying requirements for the agent-oriented paradigm. Section 4.5 provides some insights into what has been achieved in this context.

[2]That is, the goal-trees involving the scenario's goal steps.

	Type	Name	Role
1	Percept	Store_Opening	Seller
2	Goal	Send_Item_List	Seller
3	Goal	Select_Item	Buyer
4	Goal	Send_Item_Price	Seller
5	Goal	Make_Payment	Buyer
6	Goal	Validate_Card	Banker
7	Goal	Notify_Participants	Banker
8	Goal	Send_Item	Seller

Figure 4.1: Sale transaction scenario description

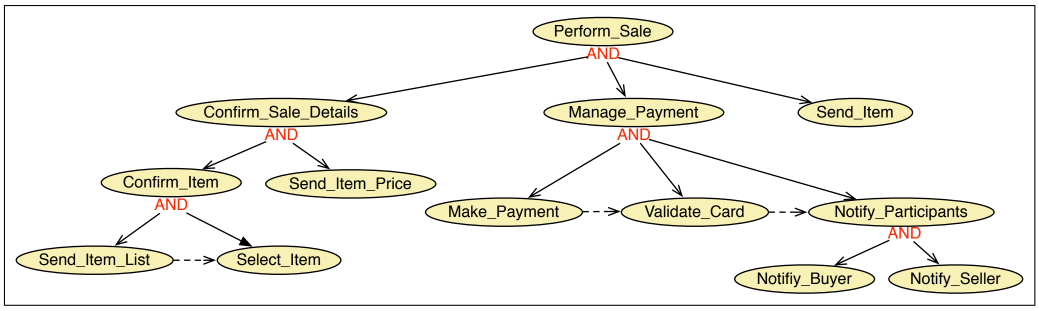


Figure 4.2: Goal-overview diagram for the trading-agent system

4.1 Constructing Activity Diagrams

In this section, we present our approach for automatically constructing an activity diagram from a scenario and a goal-overview diagram. We use the trading-agent system described[3] in what follows as a running example.

The trading-agent system models the processes that take place in a sales transaction and includes three agents: the seller, the buyer and the banker. The seller agent send the list of products to the buyer agent after receiving a ‘store opening’ percept. The buyer agent then selects a product. After that, the seller agent sends the buyer the price of the chosen item. The buyer agent may then proceed with the payment through the banker agent. The banker agent processes the payment and notifies both the seller and the buyer about the payment process outcomes (whether the transaction has been approved or denied). The order of these notifications is not important (e.g., whether—seller first and buyer second or the other way around). In the case of an approved payment, the seller must send the item to the buyer.

[3]The description here is very brief, and we refer the reader to the literature for a full description [Padgham and Winikoff 2005]

In Prometheus, system analysts translate the problem that the intended system needs to solve based on the user requirements of the system specification phase. The primary output of this phase is a set of scenarios and a goal-overview diagram (see Chapter 2, Section 2.4). The description of the trading-agent system in Figure 4.1 shows one scenario that the system-to-be needs to address and Figure 4.2 depicts the goal-overview diagram of the system.

As stated in Section 2.4, even though a scenario in Prometheus forms a single sequence of steps, there may be different ways to realise the same scenario. This is because, when there are goal steps, the goals may also be realised (and, hence, the requirements specified) through its children from the goal overview diagram. For example, the goal step *Notify Participants* in Figure 4.1 could be implemented through the step itself, the step with its children or just the children: (*Notify Buyer* and *Notify Seller* see Figure 4.2).

The construction process of the intended activity diagram involves two phases:

1. *Step-wise activity diagram generation*: this phase takes one step of a given scenario at a time and construct its equivalent activity diagram structure. Then, it concatenates the different structures to form a complete activity diagram corresponding to the specified scenario.

2. *Activity diagram reduction*: the generation phase results in an activity diagram with duplicate nodes. This phase intends to reduce these duplicates, if possible, while preserving the semantic of the original activity diagram.

4.1.1 Step-wise Activity Diagram Generation Phase

The purpose of this phase is to construct an activity diagram from the specified scenario and the relevant information to that scenario from the goal-overview diagram. This section explains the transformation of the steps in a given scenario into activity diagram control fragments.

Since the proposed approach is grounded in the Prometheus methodology, it assumes scenarios include four types of steps: actions, percepts, goals, and sub-scenarios. We transform every step into a control fragment of an activity diagram based on the type of step. The first two types (actions and percepts) are transformed into *sequential* control fragments. The goal steps are transformed into a combination of *alternative*, *parallel*, and *sequential* control fragments, depending on the decomposition of these goal steps in the goal-overview diagram. We flatten the sub-scenario steps by including those in the main scenario.

Each control fragment in the activity diagram includes action nodes that represent the steps in the scenario considered. Note that action nodes in activity diagrams model the execution of behaviour, such as operation invocations. However, we are using activity diagrams as part of requirement specification and so an activity diagram with a certain sequence of steps represents a requirement that the subsequently designed system must fulfil. Specifically, where a goal step in a scenario appears as a corresponding action node in an activity diagram, it does not denote the execution or achievement of the corresponding goal, but signifies a requirement that the designed system be able to achieve the goal. This distinction is important because the design process may end up refining the goals. For example, consider a scenario S that has a single step, $Goal1$. This yields an activity diagram with one action node in a sequential control fragment. This does not mean that $Goal1$ is executed, but indicates that the design needs to be able to achieve $Goal1$. In the case that $Goal1$ has children goals, a designer may choose to design a system that does not implement $Goal1$ directly, but rather, achieves its children.

Therefore, when mapping the goal steps in a given scenario to an activity diagram, we take into account the goal hierarchy (as depicted in the goal-overview diagram). The reason is that the process of designing a system relative to a given scenario may focus on parent or children goals of the scenario's goal steps. For example, Step 7 in the sale-transaction scenario (see Figure 4.1) is *Notify_Participants*. It is possible that the design realises this step by including the goal itself in the design. However, it is also possible that the design refines the scenario into more detail and that, instead of including the goal *Notify_Participants*, the goal's two children —*Notify_Buyer* and *Notify_Seller*—are included, which will realise the parent goal (*Notify_Participants*) as desired.

This means that, when mapping a scenario to an activity diagram, a goal step may be mapped to a process that combines the goal with its parent or children. In developing the rules for this mapping, we follow the underlying principle that *the scenario specifies design decisions and that must be honoured.*

An example is where a goal G has two children—G_1 and G_2—that are OR-refined. In this case, if we replace a scenario step G_1 with G, then we risk losing the information that the scenario designer chose to use G_1 rather than G_2.

Another example is a variant of the scenario in Figure 4.1 that replaces *Notify_Participants* with *Notify_Buyer* followed by *Notify_Seller*. The scenario specifies an order for these two goals. If we allow the design process to replace these two goals with their parent, then subsequent refinement might reintroduce the two sub-goals, but without the ordering constraint specified by the scenario, which fails to be consistent with the scenario's constraint on the order.

We now proceed to define the rules for mapping the goal steps in a scenario to an activity diagram, guided by the above principle.

Goal step merging rules

The reason for this merging process is to provide designers with various ways to realise goal steps in their design by considering the information from the goal hierarchy that is relevant to these goal steps. This information depends on the level of abstraction a designer wants their design to capture. For example, a designer may opt to realise a goal step through the goal itself. Alternatively, if the designer considers the goal step in a scenario to be of too low-level and too detailed, then they may realise the step through its parent goal. Conversely, if a designer considers the goal step to be of too high-level, then they may refine the goal and realise it in their design in terms of the goal step's descendants.

Recall that we consider a goal in an activity diagram to represent a requirement that must be realised in the subsequent design, rather than through the achievement of the goal. This means that the presence of a goal's children does not subsume the goal itself. For example, consider a goal G with three AND-refined children, G_1, G_2 and G_3. One possible design would have a plan for achieving G that makes use of sub-goals G_1 to G_3, each with their own plan.

Let us first consider the possibility of realising a goal step G in a given scenario through its descendants from the goal-overview diagram. If instead of designing (and including) G, we design its children from the goal-overview diagram, then the constraints that are captured by the scenario are not violated. To see this we consider three cases, corresponding to the decomposition type in the goal overview-diagram. There are three different decompositions captured by the goal overview-diagram (OR, SEQ and AND):

1. *Disjunctive decomposition (OR):* If a goal step G has, in the goal hierarchy, as children

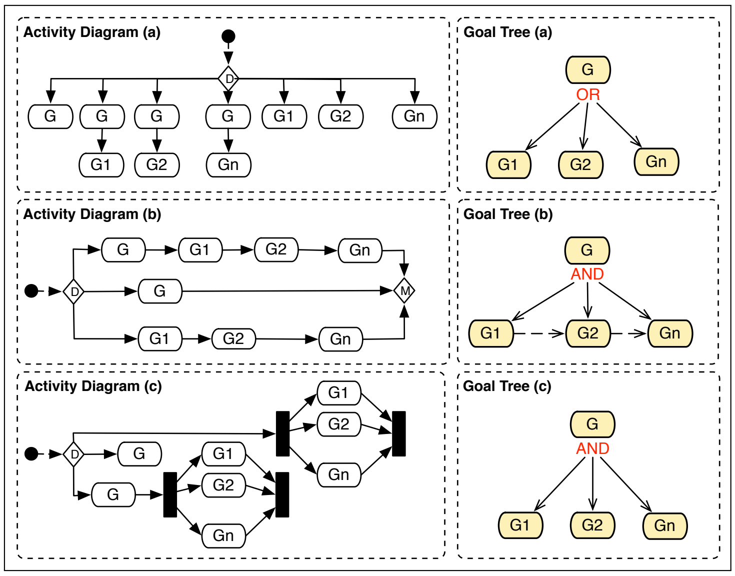


Figure 4.3: Activity diagram control fragments equivalent to the goal step transformation

G_1 OR G_2, we can realise this step by either realising G_1 or G_2 (perhaps deciding which at run-time), or we can make a design time decision to select one of them and use, for example, only design G_2. Either option is fine; if either G_1 or G_2 is included, then G is known to be realised as desired. In this case, the transformation is to an alternative fragment between[4] the goal step G, the goal step G with (any) one of its children in sequence and each of the children of the goal step G (see Figure 4.3 (a)).

2. *Directed-conjunctive decomposition (SEQ):* If a goal step G has, in the goal hierarchy as children G_1 SEQ G_2, then realising G_1 *followed by* G_2 will realise G. In this case, the transformation is to an alternative fragment between the goal step G; the goal step G with all of its children in the sequence specified; and all of its children in the sequence specified (see Figure 4.3(b)).

3. *Undirected-conjunctive decomposition (AND):* If a goal step G has, in the goal hierarchy as children, G_1 AND G_2, then a detailed design that includes the children (either in parallel or in a specified order) will realise G. In this case, the transformation is to an alternative fragment between the goal step G, the goal step G followed by all of its children as a parallel

[4]We include the option of having both the goal as well as its children because in the case where the design targets a BDI platform one typically posts the parent goal, which leads to the posting of the children (or for an OR, one of the children) goals, so both parent and children are possible.

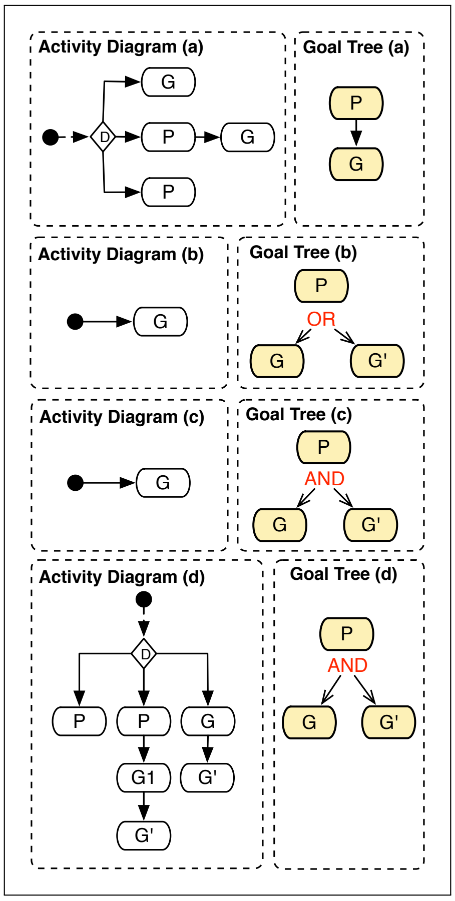


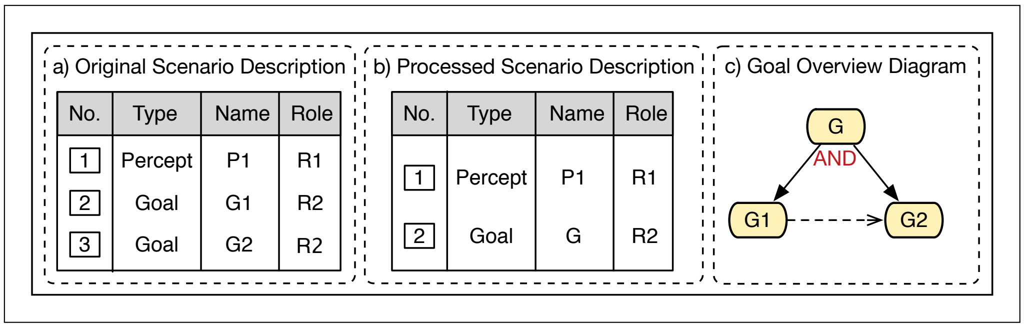
Figure 4.4: Mapping a goal step in terms of its parent

fragment and all of its children as a parallel fragment (see Figure 4.3 (c)).

We now consider the possibility of realising a goal in terms of its parent. We consider four cases: where the goal G being realised is an 'only child' (i.e., its parent P satisfies $children(P) = \langle G \rangle$), and where it has siblings, in which case there are three sub-cases, corresponding to the decomposition of the parent goal P in the goal-overview diagram (OR, AND and SEQ).

Case 1: G is an only child — In this case, if we replace G with P and the detailed design subsequently refines this replacement, then because P only has G as a child, the only possible refinement is a return to G, which is consistent with the scenario. Therefore, it is safe to replace G with P in this case (see Figure 4.4 (a)).

Case 2: G has OR siblings — In this case, if we replace the step G with P the subsequent refinement of P at the detailed design stage could choose to replace G with one of its siblings.


a) Original Scenario Description

No.	Type	Name	Role
1	Percept	P1	R1
2	Goal	G1	R2
3	Goal	G2	R2

b) Processed Scenario Description

No.	Type	Name	Role
1	Percept	P1	R1
2	Goal	G	R2

Figure 4.5: Scenario pre-processing

This is inconsistent with the scenario, since realising P implies the realisation of one of the children, which may not necessarily be the one specified in the scenario. Thus, we cannot replace G with P in this case (see Figure 4.4 (b)).

Case 3: G has AND siblings — Suppose that G has a parent P and a single sibling G' and that the scenario includes G as well as G'. If we replace G and G' with P, then subsequent refinement may end up violating the order specified in the scenario. Therefore, we cannot replace G and G' with P in this case (see Figure 4.4(c)). Note that the order specified in the scenario may not be significant: it may only be there because the scenario must specify an order. However, the point is that we do not and cannot know whether the order of G and G' is significant.

Case 4: G has SEQ siblings — Suppose that G has a parent P, which is SEQ decomposed into G followed by G'. As per Case 3, we assume that the scenario includes both G and G' as steps, and further, that G and G' appear in the scenario in an order that is consistent with their SEQ decomposition. If we replace G and G' with P, then any subsequent refinement will reintroduce G and G' in the correct order and, therefore, will be consistent with the scenario. In this case, we can allow for mapping the goal step G in terms of its parent under three conditions:

1- The scenario includes all its siblings.

2- The siblings appear contiguously in the correct order (with respect to the goal overview diagram).

3- We map all children of P as an option to design the parent goal P instead (see Figure 4.4(d)).

Since our approach processes the scenario one step at a time, we address this case by simply pre-processing the scenario, finding consecutive goals in the scenario that are SEQ siblings in the goal-overview diagram and replacing them in the scenario with their parent. Considering the scenario description in Figure 4.5 (a), 'G1' and 'G2' are consecutive goal steps. Since these two goals are only children for 'G' in the goal model in Figure 4.5 (c) and they appear in the correct order, we can replace them with their parent without violating the scenario. Figure 4.5 (b) demonstrates the processed version of the scenario. As the figure shows, both 'G1' and 'G2' were replaced by their parent goal 'G' in the goal model.

To summarise, when a goal step in a scenario is a parent goal in the goal-overview diagram, the scenario should, in some cases, include the children as *alternatives*. This is because such goal steps can be realised through the children. Similarly, under certain situations, a goal step can be realised in terms of its parent goal.

We note here that, while there are subtle variations to these cases that do indeed comply with the scenario, our aim is not to provide the designer with all possible cases, but rather to present some intuitive variations that they can work with.

Formalising The Generation Approach

We now formalise the merge and the transformation process of the goal steps in a given scenario, since the transformation of other steps is straightforward as discussed in Section 4.1.1. A scenario comprising steps $S_1 \ldots S_n$ is translated to a sequence of activity diagram control fragments, where each step is the translation of the corresponding S_i, for i *in* $1 \ldots n$.

Action and percept steps are transformed into sequential fragments. However, goal steps are transformed into different fragments, based on their different decompositions in the goal-overview diagram.

Let G be the goal corresponding to the goal step being mapped, let P denotes its parent, $children(G)$ denotes its children as a sequence of labels (where the order is the order of execution for a directed-AND decomposition, and is otherwise arbitrary) and let $decompose(G)$ denotes the decomposition type of the children of G— that is, one of the following:

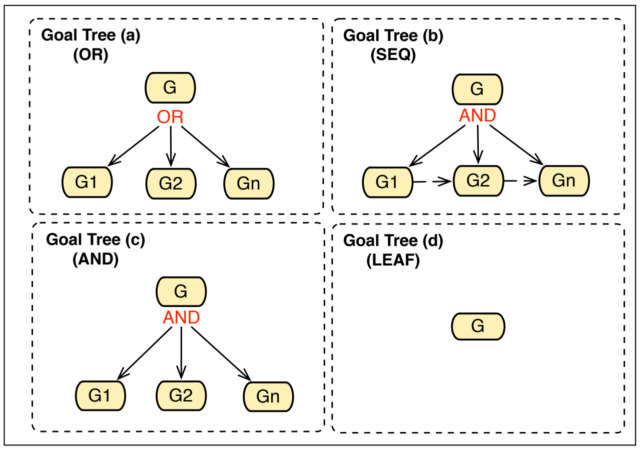


Figure 4.6: Goal tree decomposition types

1. *Disjunctive decomposition (OR):* G has children G_1 *OR*$\ldots$*OR* G_n (see Figure 4.6 (a)).

2. *Directed-conjunctive decomposition (SEQ):* G has children G_1 *SEQ*$\ldots$ *SEQ* G_n (see Figure 4.6 (b)).

3. *Undirected-conjunctive decomposition (AND):* G has children G_1 *AND*$\ldots$*AND* G_n (see Figure 4.6 (c)).

4. *Leaf:* G is a $Leaf$ if it has no children (see Figure 4.6 (d)).

For brevity we also define a simple abstract notation for depicting activity diagram control fragments: we use $seq(a_1,\ldots,a_n)$ to denote the action nodes within the activity diagram where the a_i are joined sequentially; $par(a_1,\ldots,a_n)$ to denote the action nodes within the activity diagram where all the a_i are triggered to run in parallel; and $alt(a_1,\ldots,a_n)$ to denote the action nodes within the activity diagram where there is a decision point: exactly one of the a_i is selected. Each outgoing activity edge from the decision point is guarded to ensure that only one flow has been selected. These guards are formed during the generation process based on what is included in each flow. For example, suppose the goal step G being mapped is the only child of its parent P and it has one child G'. This goal step can be realised through its parent *followed by* the goal step G *followed by* the child G', and hence the activity diagram should capture this as one of the flows that realises G. To restrict the selection of this flow, a guard such as $[Flow = Parent\ .Step\ .Child]$ should be placed on the outgoing activity edge from the decision point that belongs to that flow. It is worth noting that activity diagrams are meant to be processed by designers (human beings) and not by machines. Thus, guards are not in the formal notation.

We transform a scenario S, consisting of steps named $S_1 \dots S_n$ to the activity diagram description denoted by $seq(\widehat{S_1},\dots,\widehat{S_n})$. We use S_i to denote a step in a scenario, and $\widehat{S_i}$ to denote the corresponding action node in the activity diagram.

First, we pre-process the scenario and substitute any SEQ decomposition siblings that appear in the scenario with their parents, as discussed in Case 4 above:

$$seq(S_1,G_i,\dots G_j,S_n) \quad = \quad \begin{cases} seq(S_1,G,S_n) & \text{if } children(G) = \langle G_i,\dots,G_j\rangle \\ & \wedge\ decompose(G) = SEQ \\ seq(S_1,G_i,\dots G_j,S_n) & \text{otherwise} \end{cases}$$

in which S_i is a scenario step.

Next, we analyse each step and in some cases, realise goals with parents or children. If S_i is the name of a goal step then $\widehat{G}$ depends on the decomposition type of G, formalised as:

$$\widehat{G} \quad = \quad \begin{cases} seq(\overline{G}) & \text{if G } is\ a\ Leaf \\ alt(\overline{G},seq(\overline{G},alt(M)),alt(M)) & \text{if } DG = OR \\ alt(\overline{G},seq(\overline{G},par(M)),par(M)) & \text{if } DG = AND \\ alt(\overline{G},seq(\overline{G},seq(M)),seq(M)) & \text{if } DG = SEQ \end{cases}$$
$$\text{where } DG = decompose(G)$$
$$\text{and } \langle G_1,\dots,G_n\rangle = children(G)$$
$$\text{and } M = \widehat{G_1},\dots,\widehat{G_n}$$

We define the auxiliary function $\overline{G}$, which merges the goal itself, G, with its parent P, to obtain the sequence $seq(P,G)$ when it is permissible to do so (see the earlier discussion), and leaves all other goals as they are. The goal step G is to be merged with its parent P if—and only if—the goal step G has no siblings.

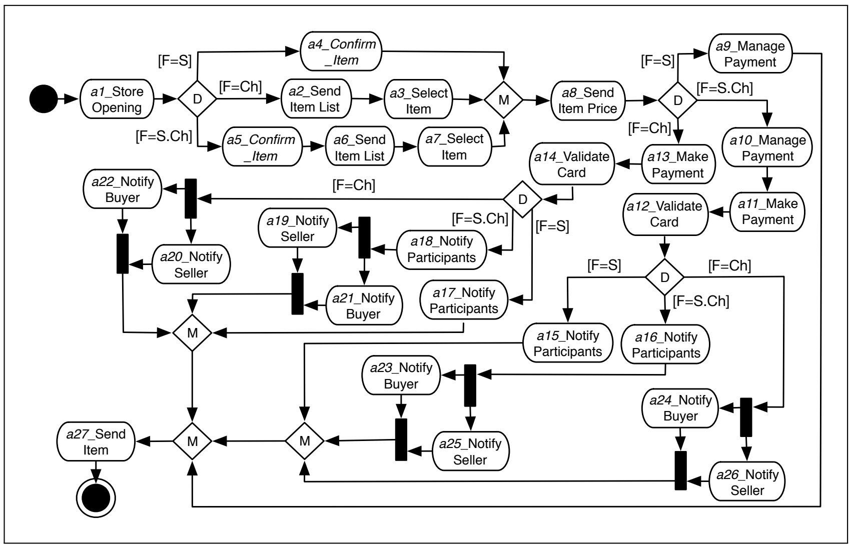


Figure 4.7: Activity diagram that merges the scenario in Figure 4.1 with the goal tree in Figure 4.2 (F:Flow, S:Goal Step, Ch: Children, P: Parent)

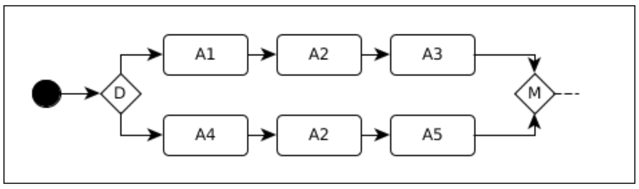


Figure 4.8: Example of a non-reducible sub-graph

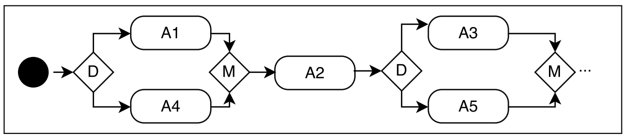


Figure 4.9: Example of merging the two repeated nodes in Figure 4.8

$$\overline{G} = \begin{cases} seq(G,P) & \text{if } children(P) = \langle G \rangle \\ G & \text{otherwise} \end{cases}$$
$$where\ P = parent(G)$$

By applying the merging rules the trading-agent system (see Figures 4.1 and 4.2) results in the activity diagram shown in Figure 4.7.

As the figure depicts, all the action nodes are prefixed with a unique identifier to ensure that the repeated nodes inhabit different incoming and outgoing vertices and are not merged. For example, consider the activity diagram in Figure 4.8. If the two $A2$ nodes are represented as a single node (see Figure 4.9) on the printed graph, then the sequence $A1, A2, A5$ would be permitted, which is not part of the intended semantics. However, this causes duplicates in the graph, which we discuss further in the following section.

4.1.2 Activity Diagram Reduction Phase

In Figure 4.7, the diagram includes several duplicate nodes, and in some cases, duplicate sub-graphs, which affects its readability. Duplicate nodes are semantically equivalent —that is, they refer to the same event, but are prefixed with unique identifiers; for example, in Figure 4.7, nodes *a19_Notify Seller* and *a20_Notify Seller* refer to the same event, but the prefixes are unique. As noted above, this is conducted so that the nodes are not merged as one node by the graph-drawing tool, which in many cases, would alter the semantics of the activity diagram. The reduction mechanism that we present merges *some* nodes (in fact, some sub-graphs) by removing the prefixes in a sound manner, such that the semantics of the original diagram are maintained.

In this section, we briefly describe how to simplify the original activity diagram generated by the merging rules to improve the readability, while maintaining the same semantics. Our approach is a set of rules to eliminate, if possible, repeated action nodes in the diagram.

The following specifies the reduction mechanism as a set of rules for activity diagrams. These rules do not reduce the nodes that have predecessors, successors or both, as that would change the semantics of the original activity diagram. For instance, the activity diagram in Figure 4.8 can not be reduced. This is because merging the two ‘A2’ nodes changes the semantics of the activity diagram as per Figure 4.9. Thus, we make no claims as to the completeness or optimality of the approach; however, the rules have worked to simplify the models on which they have been applied in our case studies.

In the following, we use the notation $S_1 \equiv S_2$ to note that two sub-graphs are semantically equivalent; that is, they refer to the same set of events, but have different prefixes on the nodes; for example, *a19_Notify Seller* $\equiv$ *a20_Notify Seller* are unique, but semantically equivalent actions.

Rule 1. Remove duplicate prefixes in alt: This rule removes duplicate sub-graphs at the start of an *alt* fragment in the activity diagram. A duplicate sub-graph prefix S_2 is removed by merging

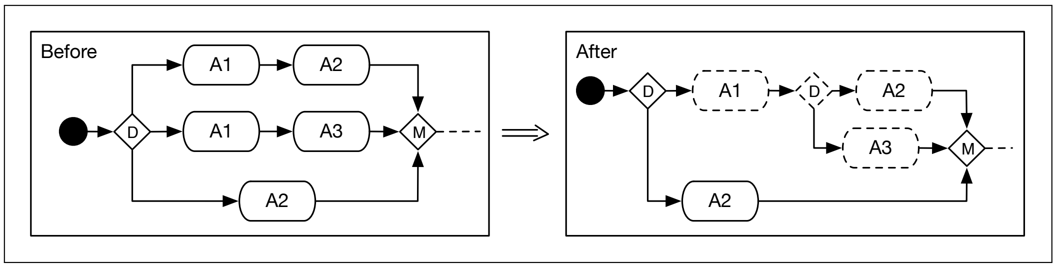


Figure 4.10: Example on activity diagram reduction (Rule 1)

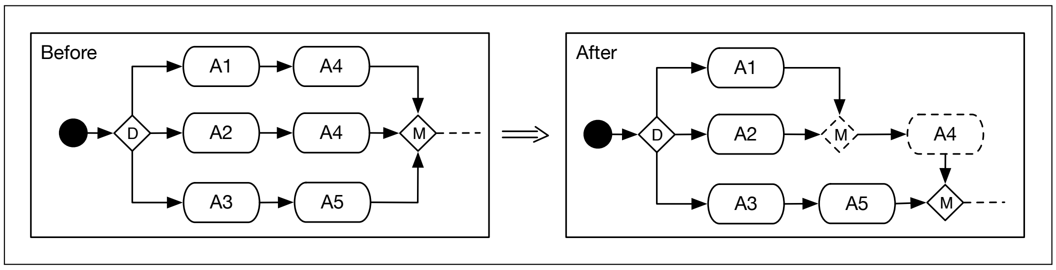


Figure 4.11: Example on activity diagram reduction (Rule 2)

the sub-graph with its semantic duplicate S_1, and then inserting an alt fragment immediately after S_1. Formally:

$$\begin{array}{lcl} alt(seq(S_1,S_m), & & alt(seq(S_1, \\ \quad seq(S_2,S_n), & \Longrightarrow & \qquad alt(S_m,S_n)), \\ \quad S_r) & & \quad S_r) \end{array}$$

in which $S_1 \equiv S_2$, and S_m, S_n, and S_r are (possibly-empty) graphs.

As an example, consider the activity diagram in Figure 4.10 with duplicate node $A1$.

Rule 2. Remove duplicate suffixes in alt: This rule removes duplicate sub-graphs at the *end* of an *alt* fragment in the activity diagram. It is essentially the reverse of Rule 1. A duplicate sub-graph suffix S_2 is removed by merging the sub-graph with its semantic duplicate S_1, and then inserting an alt fragment immediately before S_1. Formally:

$$\begin{array}{lcl} alt(seq(S_m,S_1), & & alt(seq(alt(S_m,S_n)), \\ \quad seq(S_n,S_2), & \Longrightarrow & \qquad S_1), \\ \quad S_r) & & \quad S_r) \end{array}$$

in which $S_1 \equiv S_2$, and S_m, S_n, and S_r are (possibly-empty) graphs.

As an example, consider the activity diagram in Figure 4.11 with duplicate node $A4$.

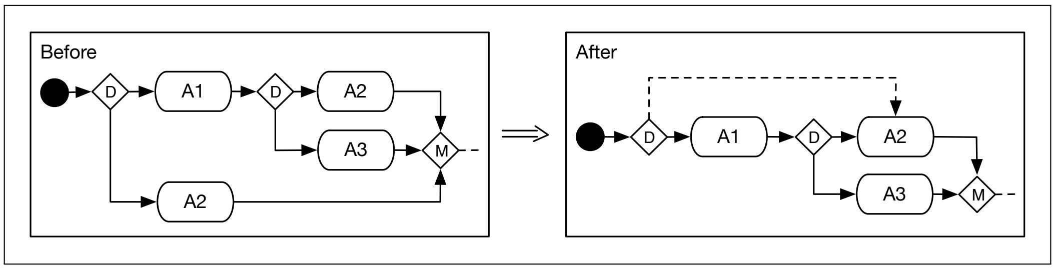


Figure 4.12: Example on activity diagram reduction (Rule 4)

Rule 3. Remove unnecessary alts: This rule removes *alt* fragments that have only one option. Formally:

$$alt(S_1) \implies S_1$$

This rule is useful after an application of Rule 1 in which all children nodes of the decision point are the same. For example, consider the example of Rule 1 above, but in which the third option containing only $A2$, was not in the original activity graph. Applying Rule 1 would result in a graph in which the first decision point had only one option: $A1$. Rule 3 would remove the unnecessary decision point.

Rule 4. Remove embedded duplicate sub-graphs: This rule removes duplicate sub-graphs (typically sequences) that occur after a decision point, with the duplicate occurring after a subsequent decision point.

$$\begin{array}{lcl} alt(seq(S_m, alt(S_1, S_n)), & & alt(seq(S_m, alt(S_1, S_n)), \\ \quad S_2, & \implies & \quad S_1, \\ \quad S_r) & & \quad S_r) \end{array}$$

in which $S_1 \equiv S_2$, and S_m, S_n, and S_r are (possibly-empty) graphs.

Note here that the only difference between the left- and right-hand sides of this rule is the replacement of S_1 with S_2. While abstractly, there are still two nodes represented, these will be drawn as a single node. As an example, consider the activity diagram in Figure 4.12 with duplicate node $A2$.

It may not be immediately obvious as to whether Rule 4 would be so useful. However, these structures occur regularly as a consequence of the OR-decomposition rule defined in Section 4.1.1. For example, as in the example above, $A1$ is a parent goal and $A2$ and $A3$ are children nodes.

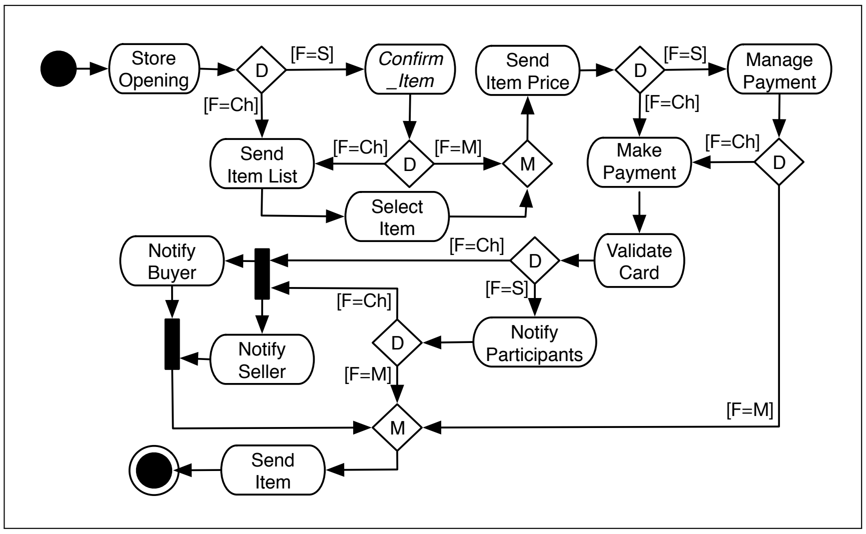


Figure 4.13: Refined activity diagram from Figure 4.7(F: flow, S: goal step, Ch: children, P: parent, M: merge)

In summary, the reduction process is performed by identifying repeated sub-graphs, and merging them by applying the aforementioned rules. For example, in Figure 4.7 the nodes $a11$ and $a6$ are followed by the portions of the graph that are identical to each other in terms of structure, and the labels of nodes (ignoring the numerical prefix on the labels). Figure 4.13 shows the refined version of the original activity diagram in Figure 4.7.

Soundness

We now prove that the four reduction rules defined above are sound. That is, they preserve the semantics of the original activity diagram. In proving this we make use of the following notations and concepts.

Given two sequences, $s = \langle s_1, \dots, s_n \rangle$ and $t = \langle t_1, \dots, t_m \rangle$ we use $s \cdot t$ to denote the result of concatenating the two sequences, i.e. $s \cdot t = \langle s_1, \dots, s_n, t_1, \dots t_m \rangle$. We extend the concatenation operator to also apply to sets of sequences: $S \cdot T = \{ s \ \cdot \ t \mid s \in S \wedge t \in T \}$. For example, let $S = \{ \langle \ s_1, \ s_2 \rangle \ \langle \ s_3, \ s_4 \rangle \}$ and $T = \{ \langle \ t_1, \ t_2 \rangle \}$. Then, $S.T = \{ \langle \ s_1, \ s_2, \ t_1, \ t_2 \rangle, \ \langle \ s_3, \ s_4, \ t_1, \ t_2 \rangle \}$.

We use $[[S]]$ to denote the meaning of S, which is the set of all traces specified by activity diagram diagram S. Specifically, $[[S]]$ is a set of sequences. Let R be a sub-graph in the activity diagram S. The activity diagram S, as explained earlier in this chapter, is specified in terms of the constructs $seq(\dots)$, $par(\dots)$ and $alt(\dots)$, and that the semantics satisfies the following two properties:

$$\begin{aligned}[\![alt(R_1,R_2)]\!] &= [\![R_1]\!]\cup[\![R_2]\!]\\ [\![seq(R_1,R_2)]\!] &= [\![R_1]\!]\cdot[\![R_2]\!]\end{aligned}$$

In other words, the semantics of $alt(\ldots)$ is the union of the semantics of each of the alternatives and the semantics of $seq(\ldots)$ is the concatenation of the semantics of the components. Since the reduction rules do not consider the parallel constructs (i.e., $par(\ldots)$), we do not need to specify $[\![par(R_1,R_2)]\!]$. We now proceed to prove soundness.

Theorem 1. *Reduction rules 1-4 are sound, ie. for each rule of the form $L \Rightarrow R$ we have that $[\![L]\!] = [\![R]\!]$.*

Proof. For Rule 1, consider the following:

$$[\![alt(seq(S_1,S_m),seq(S_2,S_n),S_r)]\!]$$
$$= \quad [\![seq(S_1,S_m)]\!] \cup [\![seq(S_2,S_n)]\!] \cup [\![S_r]\!]$$
$$= \quad ([\![S_1]\!]\cdot[\![S_m]\!]) \cup ([\![S_2]\!]\cdot[\![S_n]\!]) \cup [\![S_r]\!]$$
(since $S_1 \equiv S_2$)
$$= \quad ([\![S_1]\!]\cdot[\![S_m]\!]) \cup ([\![S_1]\!]\cdot[\![S_n]\!]) \cup [\![S_r]\!]$$
(since $(S\cdot T)\cup(S\cdot R) = S\cdot(T\cup R)$)
$$= \quad ([\![S_1]\!]\cdot([\![S_m]\!]\cup[\![S_n]\!])) \cup [\![S_r]\!]$$
$$= \quad [\![seq(S_1,alt(S_m,S_n))]\!]\cup[\![S_r]\!]$$
$$= \quad [\![alt(seq(S_1,alt(S_m,S_n)),S_r)]\!]$$

Similarly for rule 2:

$$[\![alt(seq(S_m,S_1),seq(S_n,S_2),S_r)]\!]$$
$$= \quad [\![seq(S_m,S_1)]\!] \cup [\![seq(S_n,S_2)]\!] \cup [\![S_r]\!]$$
$$= \quad ([\![S_m]\!]\cdot[\![S_1]\!]) \cup ([\![S_n]\!]\cdot[\![S_2]\!]) \cup [\![S_r]\!]$$
(since $S_1 \equiv S_2$)
$$= \quad ([\![S_m]\!]\cdot[\![S_1]\!]) \cup ([\![S_n]\!]\cdot[\![S_1]\!]) \cup [\![S_r]\!]$$
(since $(T\cdot S)\cup(R\cdot S) = (T\cup R)\cdot S$)
$$= \quad (([\![S_m]\!]\cup[\![S_n]\!])\cdot[\![S_1]\!]) \cup [\![S_r]\!]$$

$$\begin{aligned} &= [\![seq(alt(S_m,S_n),S_1)]\!] \cup [\![S_r]\!] \\ &= [\![alt(seq(alt(S_m,S_n),S_1),S_r)]\!] \end{aligned}$$

This shows soundness for Rule 2. For Rule 3 soundness is trivial, as it eliminates the alternative control fragments that have only one operand. For Rule 4 soundness is also trivial, since the only change is replacing S_2 with S_1 when $S_1 \equiv S_2$ $\square$

Prototype implementation

We have implemented the mapping from the scenario and the goal-overview diagram to the activity diagram as an Eclipse plug-in that integrates with the PDT. The tool takes the agent design file in XML format and tokenises all scenarios as well as the goal-overview diagram out of the design file. The tool applies the merging rules on the specified scenario to generate the abstract description. Then, it uses the abstract description to generate a DOT Graph source script[5], which can be used to generate a graphical depiction of the activity diagram (the Graphviz tool creates an automatic layout of nodes). We used this prototype implementation to generate the activity diagrams for the user study we conducted (see Section 4.3).

4.2 Benefits of The Activity Diagrams

The aim of the approach presented in this chapter is to provide diagrams that act as coherent structures that merge given scenarios with their related goal decomposition trees. This transformation has a number benefits besides enhancing the analysis of scenarios, since graphical notations tend to better support tasks that involve a comprehension of the overall structure [Bratthall and Wohlin 2002, Tilley and Huang 2003]. This section discusses some of the benefits of complementing scenarios and goal-overview diagrams with activity diagrams. In fact, the usefulness of supplementing the requirement specification process with activity diagrams has been studied previously [Bolloju and Sun 2012, Gross and Dörr 2009]. However, in this section we show the advantages that the activity diagrams—as a complementary artefact—have over the existing requirement specification approach of Prometheus.

[5]See http://www.graphviz.org/Documentation.php.

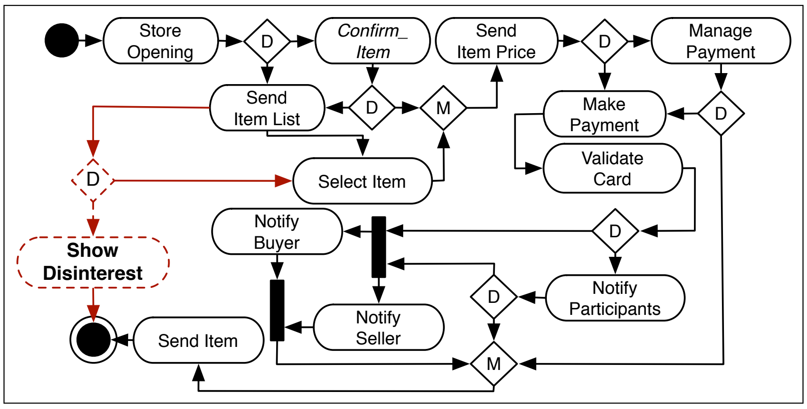


Figure 4.14: Activity diagram for the scenario in Figure 4.1 with variation

Structured representation of scenario steps and their variations

A scenario is a sequence of steps that describe a particular run of the intended system. In some cases, based on the context of a given scenario, there may be a need to go beyond a sequence of steps. Approaches for specifying scenarios in Prometheus limit the ability to include additional information, such as variations, in a structured way. For example, Prometheus scenarios do not support parallel steps, and variations are specified through natural language description. Using UML-activity diagrams provides a more structured way to visualise and specify control fragments in requirement models.

Returning to the running example, consider the following variation to the scenario in Figure 4.1: '*after the buyer agent receives the item list (step 2) from the seller agent, it may select a product (step 3) or show its disinterest*'. Thus, the flow of this scenario should branch after posting the second step ('Send Item') into two choices: (1) posting 'Select Item' or (2) posting 'Show Disinterest'. A natural linguistic description of this may faithfully capture the semantics, but interpreting this in the context of the goal-overview diagram is non-trivial, as it requires some mental effort. However, an activity diagram can capture this in a straightforward and unambiguous manner, such as the solution provided in Figure 4.14.

The existing approach (text-based) does not offer a structured way to specify parallel steps. Consider the following requirement from the trading-agent System: '*The banker agent then processes the payment and notifies both the seller and the buyer about the payment process outcomes (approved or denied)*'. Based on the specification, the banker agent needs to send notifications to both the buyer and the seller agents, but the order of posting these two notifications is not important. In Prometheus, this can be modelled by including a parent goal in the scenario and then specifying two children nodes with an undirected-AND. However, the variations introduced by these may require some mental effort from a designer to conceptualise on the part of a designer. Constructing an activity diagram to represent this through *fork/join* nodes can assist in such conceptualisation.

This structured representation of scenario steps and their variations enables our verification approach to generate a complete executable model that represents the specified scenario. The executable model will include the basic flow of the scenario, its variations and any steps that need to occur in parallel. Additionally, it will capture the possible ways to realise the goal steps in the scenario, as activity diagrams merge the goal steps with their relevant information from the goal model.

Understandability

In Prometheus, the specification of the intended system is distributed across two artefacts: the scenarios and a goal-overview diagram. The merged activity diagram provides a holistic view of how to design a given scenario, and enabling a more straightforward understanding of the behaviour of the system in the context of a particular scenario. Importantly, the activity diagram explicitly represents the possible alternate sequences of the two models. For example, the activity diagram in Figure 4.14 has seven possible execution runs after posting the 'Send Item Price' step, which requires more mental effort to ascertain from the scenario (see Figure 4.1) and the goal-overview diagram (see Figure 4.2). According to the experiments we conducted (see Section 4.4), participants were quicker at eliciting possible alternatives for a given step using the activity diagram than they were using the existing approach (the scenario and goal overview).

Maintenance

Scenarios may evolve throughout and after the analysis phase. For instance, a designer may introduce a variation that includes new goals. For example, the variation to the scenario in Figure 4.1 mentioned earlier in Section 4.2 introduces*'Show_ Disinterest'*as a new goal.

Based on the user study we conducted (see Section 4.4), incorporating newly introduced goals by considering text-based variation requires more mental effort from designers than the activity diagram. The activity diagram provides designers with a context that helps them to associate goals with existing goal trees.

4.3 User Study

In this section, we outline a controlled experimental evaluation to measure overheads and also to gauge the usefulness that a complete activity diagram has for a software engineer when analysing a Prometheus goal model and scenario. We recruited 15 participants with varying levels of experience in Prometheus and gave them several tasks to complete on simple requirements documents, while measuring aspects of their performance.

We were primarily interested in evaluating the activity diagrams for understandability—that is, how much the presence of an activity diagram impacts a person's ability to understand the requirement models—and maintainability—that is, how straightforward a design is to maintain using an activity diagram. In addition, we measured the overheads incurred by software engineers when introducing activity diagrams in the Prometheus methodology.

4.3.1 Experimental Design

We recruited a total of 15 participants in the experiment[6]. A pre-evaluation questionnaire[7] consisting of nine questions was used to measure their experience with software engineering in general, as well as with AOSE and the Prometheus methodology. All participants were familiar with intelligent agents (researchers or practitioners in a related area) and most had some experience in requirement analysis; eleven participants had experience in the Prometheus methodology and all but one had experience using some agent-oriented methodology, including Prometheus. Fourteen of the participants were experienced software developers, with the remaining participant holding a degree in computer science.

[6]Refer to Appendix A for the ethics approval.
[7]Refer to Appendix A

Participant ID	Q1.1 Requirements analysis concepts.	Q1.2 Intelligent agents concepts .	Q1.3 The Prometheus Methodology.	Q2.1 Software Analysis and Design.	Q2.2 UML Behavioural Models.	Q2.3 Use Case Scenarios.	Q2.4 BDI-Agent System Modelling	Q3 Number of software designed.	Q4 Number of agent-based systems designed.	Q5 Knowledge about other AOSE methodologies.
1	2	5	3	2	2	3	4	11-20	11-20	No
2	4	4	4	4	3	3	4	1-5	1-5	No
3	4	4	3	4	4	4	3	6-10	1-5	No
4	2	5	4	3	2	2	4	11-20	1-5	No
5	4	4	3	3	3	3	3	11-20	1-5	No
6	3	3	2	3	3	3	2	6-10	1-5	Yes
7	4	4	1	4	4	5	4	30+	11-20	Yes
8	3	4	1	4	3	4	4	30+	1-5	No
9	5	4	3	5	5	5	3	11-20	1-5	Yes
10	4	5	5	4	3	3	5	1-5	6-10	No
11	4	4	1	4	4	5	1	11-20	1-5	No
12	5	3	3	5	3	4	2	30+	1-5	No
13	4	4	4	5	5	5	3	11-20	1-5	No
14	3	4	4	4	3	2	4	1-5	1-5	No
15	5	4	3	5	4	5	3	21-29	0	No

Table 4.1: Summary of the pre-evaluation questionnaire outcomes (refer to Appendix A)

Each participant was given several tasks to perform[8]. These tasks asked participants to interpret, verify, and modify a set of requirement models for two simple systems. The presence or absence of the additional activity diagram in the requirements provided was the independent variable; this was the only difference: the remainder of the content in the requirement documents did not change. It is worth noting that the activity diagram provided was automatically generated from the scenario and the goal-overview diagram. After this, all the modifications were applied manually to the activity diagram.

[8]Refer to Appendix A

To mitigate the possibility of experience bias, for each participant, one of the requirement documents contained an activity diagram. Thus, each participant was asked to complete three tasks without the aid of an activity diagram for one system, and the same three tasks, as well as an additional fourth task, with an activity diagram for the other system (see below for a discussion of the tasks).

To mitigate the possibility of participant bias, participants were divided into two groups, G1 and G2. G1 received system S1 with an activity diagram, and system S2 without, while G2 received system S1 without an activity diagram, and S2 with an activity diagram. Given this, we had two versions for each system in the experiment.

As Table 4.1 shows, some participants were more familiar with UML behavioural models (Q2.2)[9] than Prometheus (Q1.3)[10]. To mitigate this familiarity bias, all participants were given optional material to read prior to the experiment (see Appendix A). These two pages explain the relevant parts of both Prometheus and the activity diagram to the experiment participants. Additionally, all the participants were given the opportunity to clarify any point of the optional material prior to the experiment. The aim of this evaluation was to evaluate the benefits of supplementing Prometheus with activity diagram, rather than comparing Prometheus to the use of an activity diagram.

Finally, within each group, we balanced out in which order the participants received the systems to avoid a potential bias whereby participants used their experience from the first system to answer questions on the second. That is, half of the participants received the activity diagrams with the first set of tasks and half received them with the second set.

We measured two dependent variables: time and correctness. For time we simply measured the clock time from the start to the completion of each task as a proxy for both maintainability and understandability; that is, how well does the activity diagram aid software engineers to come up to speed with the semantics of the requirement models, and to modify them. There were no time limits on tasks. For correctness, we assessed the participants' answers to each task to determine whether they had completed the task correctly. This was also used as a proxy for measuring

[9]participants were asked to rate their experience with UML behavioural models on a scale of 1-5, with 1 being inexperienced and 5 being expert.

[10]participants were asked to rate their familiarity with Prometheus on a scale of 1-5, with 1 being very unfamiliar and 5 being very familiar.

maintainability and understandability; that is, how much impact does the activity diagram have on the ability to understand and modify the requirement models correctly.

Once the tasks had been completed, participants were asked to complete a four question survey[11] asking about their experience, and their perceptions of the usefulness of activity diagrams.

4.3.2 Tasks

Each participant was asked to perform the following tasks (see Appendix A for full details):

(1) To *verify* a set of five behaviour runs—assumed to be extracted from an execution of the system—against the requirement models. The verification of a behaviour run means that its conformance to the requirement models is determined.

(2) To *specify* any three valid behaviour runs with respect to the requirement models.

(3) To *modify* a goal-overview diagram based on a new requirement proposed by a stakeholder.

(4) To *modify* an activity diagram based on a new requirement proposed by a stakeholder (only for the system for which an activity diagram was provided). The tool we developed for generating the intended activity diagram is a prototype implementation that is not stable enough for accepting digital modifications to the diagram. Thus, we asked participants to manually modify the activity diagrams (using paper and pen), as we did not want the prototype implementation to interfere with the timing results.

The purpose of the first two tasks is to measure understandability of models, while the latter two aim to measure maintainability. For example, 'Task 1' asked participants to verify whether the following valid trace of the sale-transaction system was valid indeed:

> Store_Opening, Send_Item_List, Select_Item, Send_Item_Price, Make_Payment,
> Validate_Card, Notify_Participants, Send_Item.

'Task 3' asked participants to modify the goal overview diagram given the new requirement: 'After the buyer agent receives the item list event from the seller agent, it may select a product or *may show its disinterest*'.

[11]Refer to Appendix A

		Scores (number as %)						Mean	Stdev	Mean	Stdev
Task	AD	0	1	2	3	4	5	Score	Score	Time	Time
T1	✘	0%	0%	27%	7%	40%	27%	3.67	1.18	4:16	2:49
	✔	0%	0%	0%	0%	13%	87%	4.87	0.35	2:39	1:19
T2	✘	27%	7%	27%	40%			1.8	1.26	3:04	0:48
	✔	0%	7%	7%	87%			2.8	0.56	2:50	1:01
T3	✘	67%	33%					0.33	0.49	3:08	1:38
	✔	27%	73%					0.73	0.46	2:18	1:08
T4	✔	0%	100%					1.0	0.00	1:17	1:04

Table 4.2: Task analysis results (AD = Activity diagram)

In practical terms, these tasks were small, in that each task took only a few minutes to complete. However, while small, they were not trivial—as indicated by the results, a significant proportion of the participants made mistakes.

4.4 Results

In this section, we present and discuss our experiment results.

Task analysis results

Table 4.2 presents a breakdown of the results for each task. Numbers in each cell for columns labelled zero to five represent the percentage of participants who scored that number. Scores refer to the number of traces correctly identified. 'Task 1' had five identified traces, while 'Task 2' had three identified traces; for example, for 'Task 1' performed without the activity diagram ('✘'), 27% of the participants scored two out of five. With regard to 'Task 3' and 'Task 4' a score of zero meant that the task was not achieved, while a score of one meant that the task was achieved. The mean and standard deviation for scores and completion time are presented in Table 4.2. These results show a clear trend that participants scored higher on tasks if they had the help of activity diagrams.

Understandability Recall that tasks 1 and 2 aim to measure the understandability of requirements models with and without activity diagrams. Our results provided the following observations:

The average scores for 'Task 1' and 'Task 2' are one point or more greater than those of the other tasks when aided by an activity diagram—and there are only a total of three points in 'Task 2'. Further, with the aid of an activity diagram, almost all of the participants addressed all sub-tasks correctly for 'Task 1' and 'Task 2', compared to the few participants who addressed these sub-tasks correctly without the use of an activity diagram.

Participants completed 'Task 1' on average two minutes faster with an activity diagram than without and with less variation. Completion times for 'Task 2' were slightly faster with the activity diagram.

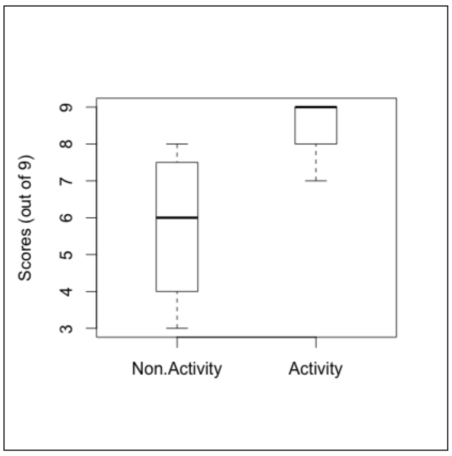
The reason for these results seems clear: with an activity diagram, the specified behaviour of the system can be easily inferred. With only a goal-overview diagram and scenario, the interplay between the two models can introduce subtle variations in behaviour that are not obvious. Adding an activity diagram helps to remove much of the ambiguity related to these variations.

However, the standard deviation in time for 'Task 1' was large as a result of one participant taking over six minutes to complete the task. Further, this participant took the longest out of all the participants in all four tasks using the activity diagram. This participant also took longer to complete 'Task 1' and 'Task 3' with an activity diagram than without, as the participant was using both the activity diagram and the scenario with the goal tree, instead of the activity diagram alone, as was required by the task. However, this participant acted as an anomaly and therefore does not indicate that the addition of an activity diagram may not always be useful.

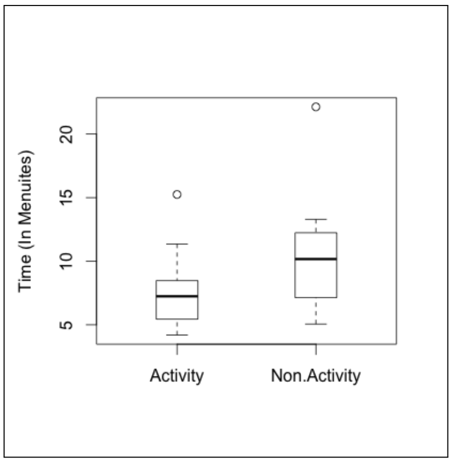
Maintainability tasks 3 and 4 aim to measure the maintainability of requirement models with and without activity diagrams. From the outcomes of the user study the following observations have been made:

For 'Task 3', the percentages and average scores show that, when aided by an activity diagram, participants were able to correctly modify the goal overview to consider new requirements more often than without the activity diagram. In addition, the average time taken to complete the tasks was 50 seconds faster with an activity diagram.

Interestingly, in 'Task 3', one participant did not even commence the task for the system without the activity diagram, despite having completed 'Task 3' with the aid of the activity diagram



(a) Scores (out of nine: “Task1” = 5 scores, “Task2” = 3 scores and “Task3” = 1 score)



(b) Time (minutes)

Figure 4.15: Box-plots for total scores and total time

previously for the other system. The participant commented that they just did not know where to start.

These results add evidence to our hypothesis that activity diagrams are useful aids when maintaining/modifying requirement models. We mostly attribute this to the fact that the participants had a solid understanding of the specified behaviour, as was also demonstrated by ‘Task 1’ and ‘Task 2’. However, ‘Task 3’ asked participants to modify the goal-overview diagram, so the results provide evidence that the activity diagram is not just useful for characterising specific behaviour traces in the requirement models, but also for considering *sets* of behaviours, which is what participants were required to do to update the goal-overview diagram.

‘Task 4’ asked participants to modify the activity diagram, so there is no comparison to be made. To assess the correctness of this, we checked whether the updated activity diagram captured the two additional traces and *only* those two additional traces. Our conclusion for ‘Task 4’ was based on the fact that all participants were able to correctly modify the activity diagram as instructed. The results here clearly support that maintaining the activity diagrams does not add a large amount of complexity to the task.

Figure 4.15(a) shows a box-plot of the scores (out of nine) for all tasks, except for ‘Task 4’. These nine scores are the total scores for the first three tasks. ‘Task4’ was excluded from this total, as it is about maintaining activity diagrams, and hence, it is not in the *non-activity* version.

Qst	SD	D	N	A	SA	Mean	Stdev
Q1	0%	0%	7%	20%	73%	4.67	0.62
Q2	0%	0%	0%	53%	47%	4.47	0.52
Q3	0%	0%	0%	47%	53%	4.53	0.52
Q4	0%	0%	0%	40%	60%	4.6	0.51

Table 4.3: Post-evaluation questionnaire (Responses: [SD] Strongly Disagree=1, [D] Disagree=2, [N] Neither=3, [A] Agree=4, [SA] Strongly Agree=5), see Section 4.3.1 for the questions.

These plots demonstrate that participants did better in the activity-based approach with a median value equals to nine (the maximum score), compared with a median of six for the non-activity approach. Further, the minimum score for the activity-based approach is greater than the median score for the non-activity approach ($7 > 6$). With respect to the time taken for each approach (see Figure 4.15(b)), the activity-based approach takes less time than the non-activity approach. All the minimum, median and maximum values (4:37, 7 and 15:25 respectively) of the time taken to complete the activity-based approach are less than those for non-activity approach (5:06, 10:30 and 22:14 respectively).

Post-evaluation questionnaire

Table 4.3 shows the results for the post-evaluation questionnaire. We asked participants to rank their experience by agreeing or disagreeing with the following four statements:

Q1. The activity diagram enables an easy extraction of a possible behaviour path relative to a particular scenario.

Q2. The time taken to grasp what the entire activity diagram shows is reasonable.

Q3. It is easy to maintain activity diagrams (and easy to incorporate changes including: adding, removing and modifying entities).

Q4. Activity diagrams would be useful in designing the agents for a particular scenario.

The results here demonstrate a strong preference for the inclusion of activity diagrams; they show that participants felt the diagrams were useful for understanding behaviour, were straightforward to use, and would aid with the design process.

In addition to the four statements above, we asked participants to answer an *open-ended* question. The question asked the study participants about the approach they most preferred and to

	Activity Diagam		Non-Activity	
Participant_ID	Scores (out of 9)	Time (in minutes)	Scores (out of 9)	Time (in minutes)
1	9	15:25	8	8:15
2	9	7:33	4	11:35
3	9	8:41	8	22:14
4	8	8:35	4	5:06
5	8	11:36	7	13:30
6	9	11:34	6	13:33
7	8	5:10	3	7:00
8	9	4:37	8	10:17
9	9	5:38	4	12:15
10	7	5:51	8	7:27
11	9	7:21	5	12:33
12	9	7:24	3	6:24
13	8	4:19	7	10:04
14	7	8:54	5	10:17
15	8	6:05	7	6:49
Mean	8.4	7:55	5.8	10:29
CI	0.373	1:41	0.96	2:20
Upper confidence bound	8.77	9:36	6.76	12:49
Lower confidence bound	8.02	6:13	4:84	8:08

Table 4.4: Summary of the total results for each participant in both approaches (CI: confidence interval value at a confidence level of 95%)

justify why preferred it. The answers to the open-ended question also further confirm the results obtained from the post-evaluation questionnaire in Table 4.3:

> "*The approach with the activity diagram is preferable. While I prefer (I think) to design agents using the goal overview, analysing possible interactions between agents, sequence of actions etc. is made easier with the activity diagram. Design flow would be easier to find through analysis of activity diagram, as you have a single line to follow rather than multiple goal trees.*" —Study Participant 1.

Some participants noted that it was not the activity diagram itself that was useful, but the activity diagram in combination with the goal overview-diagram:

> "*The one with activity diagram is more helpful, but it is not a substitute for goals. It just makes the trace analysis easier because the sequence of events is integrated*

> *regardless of the agent by whom they are caused.*" —Study Participant 7.

Three other participants noted that the usefulness of activity diagrams is not limited to understanding behaviour; it may also facilitate the communication between the system analyst and the stakeholders.

> "*I would prefer to use the approach that uses activity diagram, because it was very useful to understand the behaviour of the system in simple way that could help the requirements analyst to share it with the stakeholders which will help to mitigate the problem of communicating the requirements with clients.*" —Study Participant 12.

Statistical Significance

The two measures taken were the (*correctness* and *time*). Table 4.4 lists the scores, times, and the means of the scores (out of 9) as well as the time taken for all the participants in the first three tasks using each approach and the confidence intervals at 95%. Note that we excluded the fourth task using the *activity-based* approach from this comparison, as it does not have an equivalent task using the other approach.

Given the estimated range of the scores of the activity-based approach ($8.02 \leq mean \leq 8.77$) and the non-activity approach ($4.84 \leq mean \leq 6.76$), it is noted that these intervals do not overlap; therefore, we can conclude that the results for the scores are significant with a 95% level of confidence.

It is clear that the participants took less time using the activity-based approach on average, compared to when using the non-activity approach. However, the confidence intervals for these samples overlap, so further analysis is required.

The sample under analysis is a paired sample, as all 15 participants undertook both approaches. Using a Shapiro test [Royston 1995], we concluded that the data was not normally distributed. Thus, we used the *Wilcoxon-signed-rank* non-parametric test [Hollander et al. 2013] for our samples. The null hypothesis (H_0) is that there is no difference between the times in the activity-based approach and the non-activity approach, while the alternative (H_1) is that there is a significant difference.

We ran the test in *R* with a significance level of 95% ($\alpha = 0.05$). The resulting *p*-value was 0.044 (p-value ≤ 0.05), so we rejected the null hypotheses (H_0). Thus, there is a *significant* difference between the times in both approaches.

Threats to validity

The main threat to the external validity of our experiment is the *scale* of the designs used. The size of the system used in the study was comparable to those in industry-level projects. In fact, it was larger than the systems in the industry-level projects used for our evaluation in Chapter 7. We did not use industry-level projects in this study because they are not as easy to understand compared with the system we have employed. However, much larger systems may result in a more complicated activity diagram that requires more time to understand than the original approach. Thus, further experimentation on larger systems is needed.

Since the first two tasks require participants to deal with behavioural runs, *maturation* is an internal threat to the validity of the experiments. The first task asks participants to validate behavioural runs, while 'Task 2' requires them to specify possible behavioural runs given the same scenario and goal tree. Thus, participants may use their experience from 'Task 1' to achieve 'Task 2' and, hence, the correctness of and the time taken to complete 'Task 2' may be affected. As a result, the experiments need to distinguish 'Task 1' from 'Task 2', by either varying the goal tree or the scenario to obtain more reliable results. However, we distinguished these two tasks by rearranging the choices (see appendix A). We observed that all participants tackled both tasks independently (i.e., they did not seem to rely on their experience from 'Task 1' in achieving 'Task 2'). In fact, all participants specified different behavioural runs in 'Task 2' than those used for 'Task 1'.

4.5 Related Work

In the context of AOSE, there has been a strong focus on models for requirements, but little research into the relationships between these models. The six AOSE methodologies mentioned in Chapter 2 each supports the requirement gathering and analysis process. Further, most of the methodologies share similar requirement specification elements [DeLoach et al. 2009]. Such processes result in a variety of artefacts and, hence, the information is scattered across these artefacts. Despite the fact that these methodologies offer development environments through their supported

	Requirements via Multiple Artefacts.	*Structured Representation for Variations.*	*Use of UML activity diagrams.*	*Produces holistic view of various requirements artefacts.*	*Comments.*
Prometheus	✔	✘	✘	✘	There is an attempt for structured specification of variations ([Thangarajah et al. 2011])
Tropos	✔	✘	✘	✘	There is a proposal to transform the early requiremnts model in the secure Tropos into use-case models ([Alam et al. 2015])
MaSE	✔	✘	✘	✘	—
INGENIAS	✔	✘	✘	✘	—
Gaia	✔	✘	✘	✘	there is an attempt to extend the methodology through in- corporating the AUML modelling notations including sequence diagram and activity diagrams ([Duran-Faundez et al. 2015])
ROADMAP	✔	**P**	✘	✘	—
PASSI	✔	✘	✔	✘	Activity diagrams are used in specifying the capabilities of each agent, and not to specify the requirements of the system-to-be.
Prometheus + Our Approach	✔	✔	✔	✔	Our approach complements the requirements specification process in the Prometheus methodology.

Table 4.5: Features supported by the seven AOSE methodologies considered in this thesis concerning requirements specification (✔: feature is supported, ✘: feature is not supported, **P**: feature is partially supported)

tools, they do not offer the ability to merge information from the different artefacts into one cohesive structure (see Table 4.5). Further, some of them, such as Prometheus and ROADMAP do not provide a structured approach for specifying variations to the requirements, while other, such as MaSE, do provide such an approach.

In [Al-Hashel et al. 2007], a comparison between three methodologies (ROADMAP, Prometheus, and MaSE) is conducted. The comparison shows that all these methodologies share the same notions of goal hierarchy and use cases when specifying requirements. However, they provide little, if any, support for the structured specification of variations or for merging different artefacts into a single representation. As shown in our evaluation, this single representation enhances the capacity for understanding and maintaining the requirements.

MaSE uses sequence diagrams to represent the textual use-case scenarios. However, the methodology does not consider the goal hierarchy in this process [DeLoach et al. 2001]. More importantly, the tool support (the agent tool [DeLoach 2001]) of the methodology does not automatically generate the sequence diagram equivalent to a given scenario that may be error prone.

There has been an attempt, extending Prometheus, to enhance the approach to specifying requirements. In [Thangarajah et al. 2011], the authors proposed a more structured way for specifying requirements in Prometheus. Their focus was on generating scenario-based test cases for the run-time testing of agent systems. They provided a means for specifying variations in terms of actions and percepts and showed how scenarios may be traced to the detailed design of the agents for coverage. Future work would involve investigating a similar traceability approach to the activity diagram based approach presented in this work.

the scenarios of ROADMAP/AOR [Sterling and Taveter 2009] already support the structures considered in our work, including sequences, branching and parallelism. The notation used is tabular rather than graphical, but it can be used to specify traces in a similar ways to activity diagrams. However, the methodology does not support the concept of a single coherent structure representing both scenarios and goal models.

With respect to Gaia, there has been an attempt to extend the methodology by incorporating the AUML modelling notations, including by the use of sequence diagrams and activity diagrams [Duran-Faundez et al. 2015], which has been presented through the development of a flexible manufacturing control system following the Gaia methodology. As stated in Chapter 2, in Gaia the roles (functions) of the intended system are specified as role models forming an initial step. Then, the interaction models are used to model the interactions between agents and the dependency between these different roles. In [Duran-Faundez et al. 2015], AUML-sequence diagrams are used to model the interactions. However, these diagrams are manually generated. Further, the authors use activity diagrams to manually model the internals of each agent based on its functions (*roles*).

Alam et al. in [Alam et al. 2015] propose guidelines by which to transform the early requirement models of the secure Tropos into UMLsec[12] use-case diagrams. Such transformation is meant to show the dependency between the actors and the functionalities that require security in the intended system. Thus, they provide a mechanism to elicit information from a larger model (an early requirement model) into a cohesive model (a use-case diagram). They use the Kent Modelling Transformation Language (KMTL) for the transformation; however, they do not indicate whether this transformation is automatic or not. In addition, their proposed approach does not allow designers to edit such models, and incorporate variations.

There is other relevant work on automating the transformation of use-cases into UML-activity diagrams. Yue et al. [Yue et al. 2010] propose a sophisticated algorithm to automatically transform use cases into UML-activity diagrams. The use cases supported in their approach must follow a tabular format that includes the basic and the alternative flows. Yue et al. [Yue et al. 2010] adopt different natural language processing techniques for analysing the free-text description of the use cases. Thus, the use cases description must follow the RUCM template that has 26 well-defined restriction rules [Yue et al. 2009]. However, the proposal in [Yue et al. 2010] does not consider other artefacts than use cases, whereas our approach considers goal hierarchy.

Gutirrez et al. [Gutiérrez et al. 2008], developed an approach to automate the process of generating an activity diagram out of a use-case scenario. They use the QVT-rational language (*query/view/transformation*) for the model transformation and provide designers with a meta-model that must be followed when specifying use-case scenarios. This meta-model allows designers to specify the core elements of the intended scenario, including participants, pre-conditions, post-conditions, the main flow of steps and any alternatives. Even though the meta-model provided in their work is similar to the structure of scenarios in Prometheus, it does not allow for the specification of parallel steps, which is supported by Prometheus. Scenarios can implicitly capture parallelism through having a goal step in the scenario that has children of the *undirected-conjunctive* decomposition. Unlike our approach, the meta-model does not allow designers to merge any relevant information to the specified use case scenario from other artefacts.

[12]UMLsec is an extension to UML notation to enable the development of secure systems [Jürjens 2002].

Summary

In this chapter, we have proposed an approach that complements the current approach for specifying requirements in the Prometheus methodology. The approach is to introduce activity diagrams to be used as coherent structures that merge the steps of a given scenario along with their related information from a goal-overview diagram. Although our approach is grounded in Prometheus, it is applicable to any methodology that link the notions of goal hierarchies and use case scenarios in the system specification phase.

We discussed the benefits our approach offers designers with respect to different design aspects and how it overcomes the limitations of the current approach of Prometheus. Our method enables designers to specify more control fragments, such as parallelism and variations in a more structured manner. Since we have a better structured representation, we can ensure that the executable model that represents the specified scenario covers all possible flows in the scenario, including variations. For example, we transform activity diagrams in our approach, as discussed in Chapter 5, into Petri nets.

Our user study showed that including activity diagrams leads to a better understanding of the specification by the provision of a more holistic view on what the intended system is meant to achieve and how it should behave and that it provides assistance when performing maintenance on the system. More importantly, the results showed that the inclusion of activity diagrams as an extra artefact does not have any significant overhead for designers. Participants in our experiment unanimously agreed that the inclusion of the activity diagram improved their abilities to understand the requirement models.

CHAPTER 5

Static Models to Executable Models

Protocols in Prometheus are specified using AUML notation, namely AUML-sequence diagrams. Similarly, scenarios are captured via structures akin to use case structures in the object-oriented paradigm, with goals that are modelled using a goal model. Agents in Prometheus are designed in terms of plans that are associated with each other through different entities (e.g., events, actions and percepts). Such designs are considered semi-formal models. With this in mind, it is necessary to transform these models into more general and executable models with precise semantics so they can be used by our automated checking approach.

This chapter explains the technical aspects of transforming the semi-formal models in Prometheus to executable models. These models are used for ensuring the conformity of all possible behaviour runs from the agent-behaviour models with specified point-of-reference artefacts (i.e., scenarios and interaction protocols). We have choosen the simple *place/transition* Petri nets (see Section 2.6) to act as executable models, as they can model the essential control fragments (i.e., sequential, selection, loop and parallel) that are captured by both the point-of-reference artefacts and the agent-behaviour models. Section 5.1 details the transformation of the point-of-reference artefacts: (1) scenarios with a goal model (i.e., an activity diagram from Chapter 4) and (2) protocols. Section 5.2 explains the transition from agent-behaviour models to executable models with the aim of extracting all possible behaviour runs.

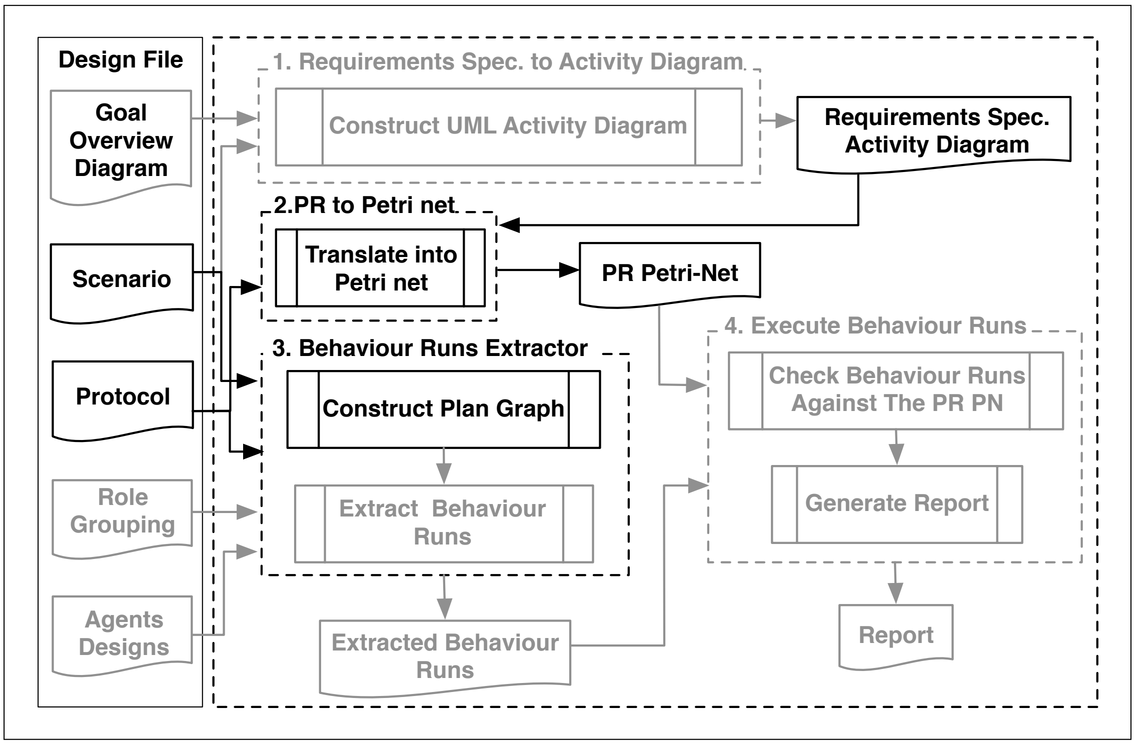


Figure 5.1: Static verification framework (PR: *point-of-reference* artefact, PN: Petri net)

5.1 Executable Reference Models

Figure 5.1 outlines the modules of our verification framework. In this section, we concentrate on the module that translates the point-of-reference artefacts to Petri nets (labelled *'2.PR to Petri net'* in Figure 5.1). As per Figure 5.1, the input to this module is either a protocol or an activity diagram that merges a scenario with its relevant information from the goal overview diagram (see Chapter 4). The module results in a *place/transition* Petri net that serves as an executable model to the processed point-of-reference artefact (i.e., an activity diagram or a protocol).

Although there are many formalisms, such as statecharts [Harel 1987], which provide richer semantics than Petri nets, we opt for Petri nets since they are simple to understand, and they fulfil the needs of our approach. Further, there are existing algorithms proposed by Poutakidis et al. that translate AUML-sequence diagrams into Petri nets [Padgham et al. 2005b].

Both activity diagrams and AUML-sequence diagrams may capture a single control fragment or a combination of control fragments. There are five different control fragments that a point-of-reference artefact can depict (see Section 2.3.2 and 2.7): *sequential* (SEQ), *selection* (ALT),

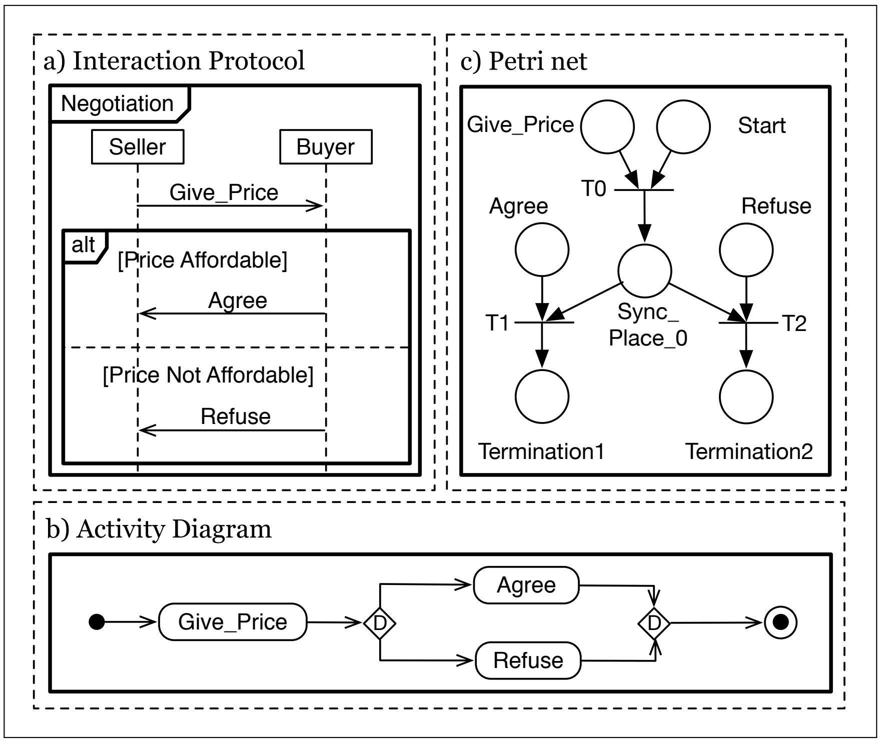


Figure 5.2: An interaction protocol and its equivalent activity diagram with their Petri net

optional (OPT), *parallel* (PAR) and *loop* (LOOP). Each control fragment is then transformed to its equivalent *place/transition* Petri net fragment. The concatenation of all the Petri net fragments results in a net that represents the specified point-of-reference artefact.

For example, Figure 5.2 (a) depicts an AUML-sequence diagram that describes the interaction between two agents. Briefly, the protocol organises a negotiation process between the ‘Seller’ and the ‘Buyer’ agents. The ‘Seller’ agent gives the price to the ‘Buyer‘ agent. Then the ‘Buyer’ agent alternatively either agrees on the given price or refuses it. The protocol captures two control fragments: sequential and selection.

The AUML-sequence diagram in Figure 5.2 (a) is textually described as $SEQ\ (Give\_Price,$ $ALT\,(Agree,\ Refuse))$. The activity diagram in Figure 5.2 (b) models the same protocol. Similar to the AUML-sequence diagram, the activity diagram is textually described as $SEQ\ (Give\_Price,\ ALT$ $(Agree,\ Refuse))$. Since both AUML-sequence diagrams and activity diagrams capture the same control fragments and they both share the same textual description, their transformation to Petri nets is the same from an abstract perspective.

Figure 5.2 (c) shows the equivalent Petri net that defines the semantics of the negotiation

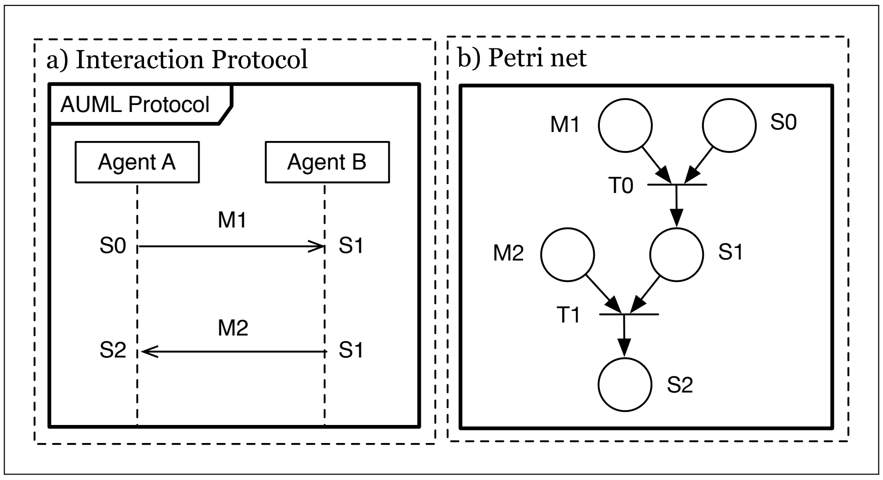


Figure 5.3: Simple AUML protocol with its states and its Petri net

protocol described above. The Petri net precisely matches the semantics of the AUML-sequence diagram in Figure 5.2 (a) and in the activity diagram in Figure 5.2 (b). The Petri net captures two fragments: sequential and selection. The 'T0' transition represents the sequential control fragment, while the transitions 'T1' and 'T2' depict the selection control fragment. After the 'T0' transition fires, the 'Sync_Place0' place will have a token, allowing only one of the transitions 'T1' and 'T2' to fire. The 'Sync_Place0' place concatenates the two fragments to form the full Petri net equivalent to the AUML-sequence diagram and the activity diagram (i.e., an interaction protocol and a scenario). The 'Start' place is a special place that indicates the *initial state* of the Petri net.

5.1.1 Point-of-Reference Artefacts to Petri Nets

Poutakidis et al. [Padgham et al. 2005b] have developed a debugging framework that ensures the correctness of the interactions between Prometheus-based agents. The debugging is performed by monitoring the system while it is in its running state. The framework compares the agent-behaviours with the expected behaviours based on the design artefacts (i.e., the interaction protocols) and then reports any discrepancies. Thus, the framework needs to process the design artefacts to produce debugging components. Converting design artefacts to debugging components means generating suitable machine interpreted components from the design artefacts.

The debugging approach of Poutakidis et al. uses AUML-sequence diagrams for mod-

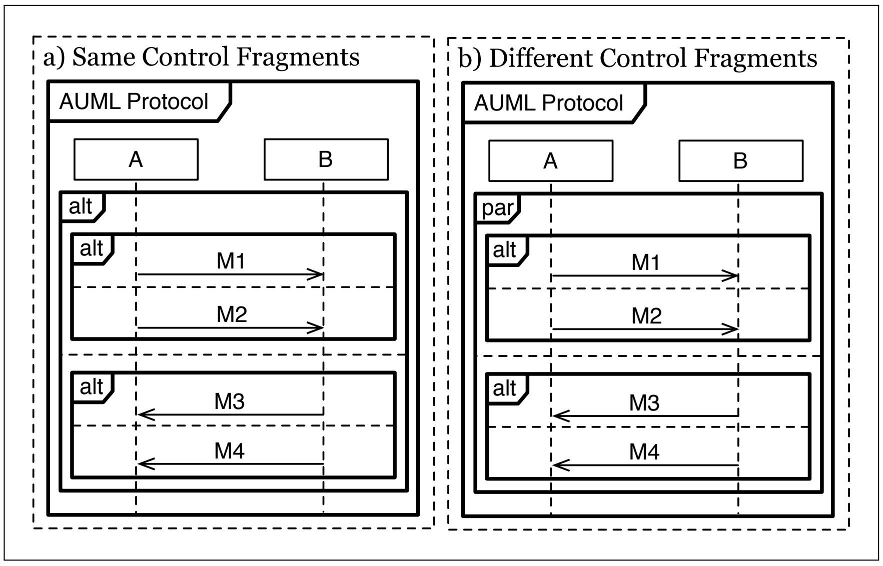


Figure 5.4: Interaction protocols with nested structures

elling interaction protocols. AUML-sequence diagrams are not precise enough to be interpreted by machines. Hence, Poutakidis et al. have chosen to translate AUML-sequence diagrams to *place/transition* Petri nets.

Poutakidis et al. have developed a translation algorithm for each control fragment of the AUML-sequence diagram. It is important to note that *we directly adopt* these translation algorithms in our approach. However, we modify the transformation of the parallel control fragment as detailed in Section 5.1.2.

The translation considers an AUML protocol as a set of interaction states between participants. An interaction occurrence (e.g., a message being sent to another agent) causes the protocol to advance from one state to another. For example, by posting 'M1' in Figure 5.3, the protocol advances from state 'S0' to state 'S1'. The protocol then progresses towards its final state 'S2' by sending 'M2'. These transitions between the protocol states are modelled as a Petri net (see Figure 5.3 (b)). Initially, the 'S0' place of the Petri net would have a token to indicate the beginning of the protocol. The posting of the 'M1' message is indicated by placing a token into its corresponding place in the Petri net. Then, the transition 'T0' is enabled and can be fired, as all its input places have tokens. By firing the 'T0' transition, the Petri net advances to the 'S1' state, which indicates an advancement from the 'S0' state to the 'S1' state in its corresponding protocol in Figure 5.3 (a).

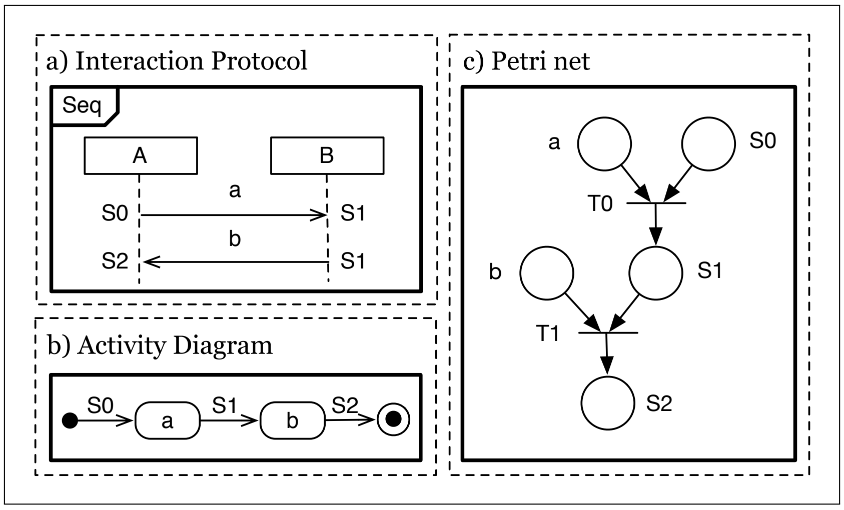


Figure 5.5: A sequential control fragment with its Petri net

Note that AUML protocols can capture the *nested* structures of many control fragments. An example of a nested structure is a selection control fragment inside another selection control fragment (see Figure 5.4 (a)). Such structures are not necessarily of the same type (e.g., a selection within a selection) they may also be of different types (e.g., a selection within a parallel), as per Figure 5.4 (b).

In the following, we describe the translation approach to the control fragments captured by both the AUML-sequence diagrams and the UML-activity diagrams:[1]

- *Sequential* (*SEQ*): the sequential control fragment indicates that the execution order of the entities of the specified point-of-reference artefact (i.e., protocol or scenario) is by one entity (i.e., a step or a message) after another. Figure 5.5 illustrates the sequential control fragment and its equivalent Petri net. The protocol in Figure 5.5 (a) shows that the ‘a’ message must be sent before the ‘b’ message and that both messages must be sent to satisfy the protocol. Similarly, the activity diagram in Figure 5.5 (b) states that the ‘a’ step must be realised before the ‘b’ step and that both steps must be realised before the activity diagram reaches its final state.

  The Petri net version of both the AUML protocol and the UML-activity diagram is shown in Figure 5.5 (c). As can be seen from the figure, the Petri net enforces the execution flow

[1]The control fragments listed are shared in modelling protocols and scenarios. However, we refer the reader to ([Poutakidis 2008], Chapter 4) for a comprehensive list of the AUML-sequence diagram control fragments.

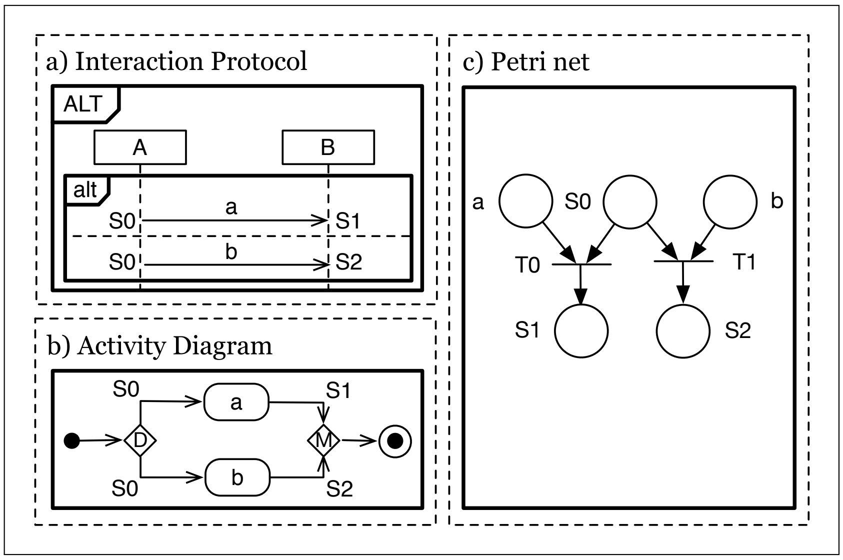


Figure 5.6: A selection control fragment with its Petri net

captured by the behaviour diagrams in Figure 5.5 (a) and Figure 5.5 (b). The Petri net will not reach its termination place (‘S2’) unless both the ‘T0’ and ‘T1’ transitions are fired in sequence. Given this, place ‘a’ must receive a token first to fire the ‘T0’ transition and then place ‘b’ should receive a token to enable the ‘T1’ transition to fire.

- *Selection* (*ALT*): this control fragment is used to model alternative ways to realise the same goal (i.e., a step or a communication). The protocol in Figure 5.6 (a) captures a selection control fragment with two regions, where only one region is to be executed. As a result, only one message out of the two messages (‘a’ and ‘b’) will be sent. Likewise, the activity diagram in Figure 5.6 (b) starts with a decision point that splits the execution flow into two branches, with only one branch to be executed (i.e., either the ‘a’ step or the ‘b’ step will be performed).

  The Petri net equivalent to the behaviour diagrams in Figures 5.6 (a) and 5.6 (b) is shown in Figure 5.6 (c). As per the figure, the Petri net begins from state ‘S0’ to allow only one transition to be fired (either ‘T0’ or ‘T1’). If the ‘a’ entity occurs, then the Petri net will reach the ‘S1’ termination place. Alternatively, if the ‘b’ entity occurs, then the termination place ‘S2’ will be reached.

- *Optional* (*OPT*): in this control fragment, the entities (i.e., the steps or messages) within

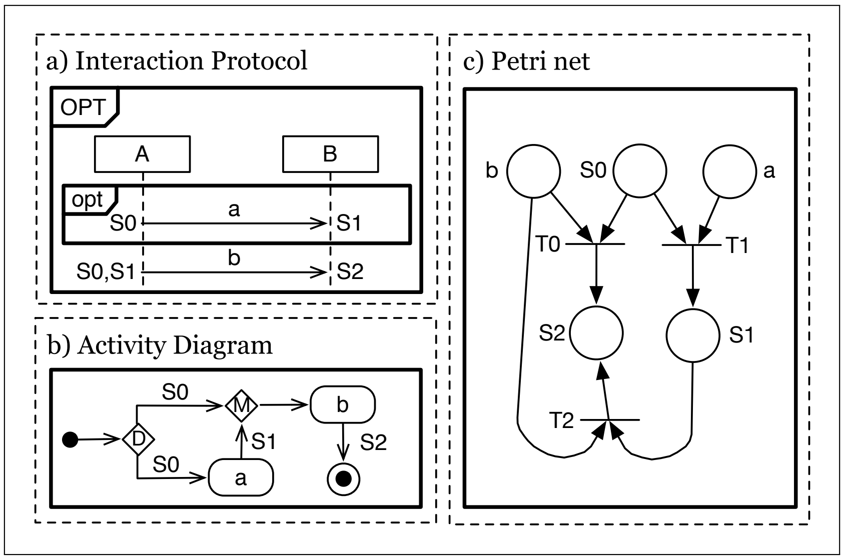


Figure 5.7: An optional control fragment with its Petri net

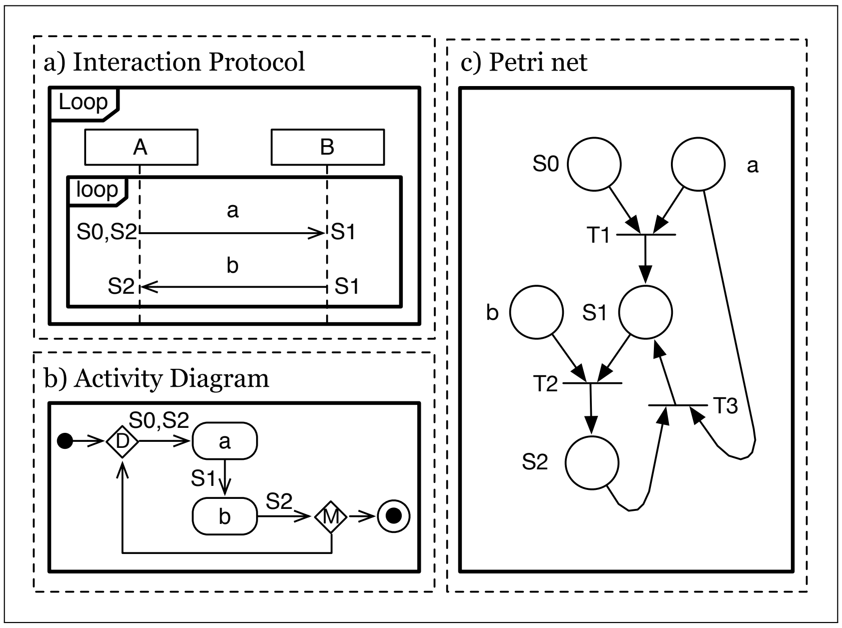


Figure 5.8: A loop control fragment with its Petri net

the fragment may or may not occur. The protocol in Figure 5.7 (a) has an optional control fragment with only one message to be sent: ‘a’. Similarly, the activity diagram in Figure 5.7 (b) has a decision point with two branches. One branch indicates the ‘a’ step, while the other directs the execution flow towards the merge point that combines the branches and to the termination state of the activity diagram.

The Petri net equivalent to the behaviour diagrams in Figures 5.7 (a) and 5.7 (b) is shown in

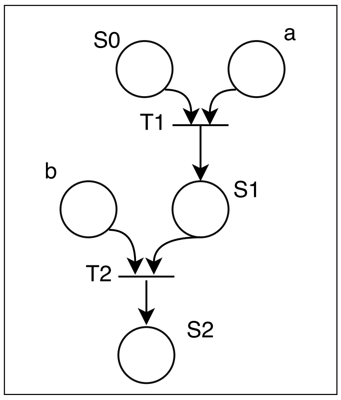


Figure 5.9: Petri net for a loop control fragment with one iteration

Figure 5.7 (c). As per the figure, the Petri net begins with the 'S0' state, allowing either the 'T0' or the 'T1' transition to be executed. The execution of the 'T0' transition means that the optional control fragment is executed, whereas firing the 'T1' transition means that the optional control fragment is discarded.

- *Loop* (*LOOP*): the loop-control fragment enables its enclosed entities (i.e., its steps or messages) to occur multiple times. Since our approach considers a static view of the system-to-be, it cannot determine the number of iterations a loop-control fragment undergoes. Thus, we assume that a loop control fragment iterates only *once* (see Assumption 3 on page 63).

  The AUML protocol in Figure 5.8 (a) enforces that the messages 'a' and 'b' can be sent multiple times. In Figure 5.8 (b), the activity diagram states that steps 'a' and 'b' can be executed multiple times.

  The Petri net version, based on Poutakidis' transformation approach to the loop-control fragment is shown in Figure 5.8 (c). The Petri net starts with the 'T1' transition and ends with the 'T2' transition. However, the Petri net also has the 'T3' transition that enables the execution of the Petri net multiple times, as 'T3' can be treated as 'T1'. This is because the termination place of the Petri net ('S2') is linked to the 'T3' transition of the 'a' place. For example, the following sequence of entities would result into two iterations: $\langle\, a,\, b,\, a,\, b \,\rangle$. However, recall that we treat loops as a cycle with one iteration; therefore, these two entities (i.e., 'a' and 'b') should occur only once. Therefore, we eliminate the 'T3' transition, which consequently eliminates the repetitions (see Figure 5.9).

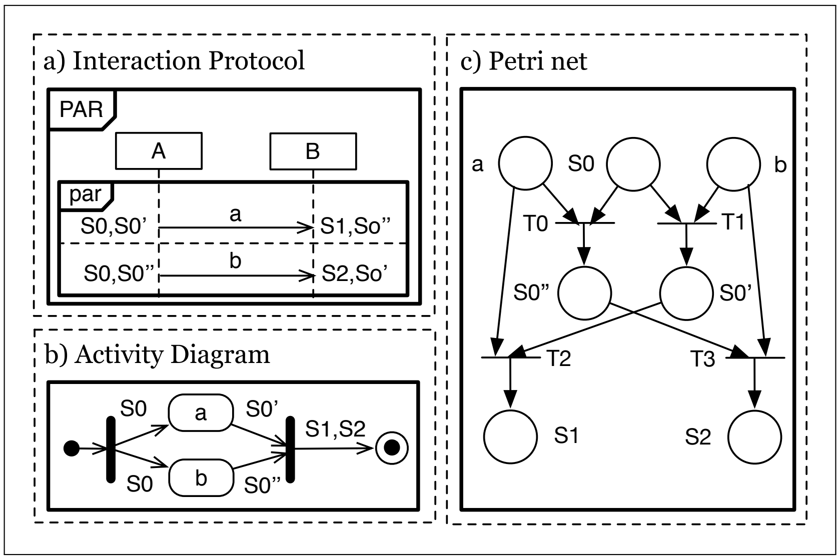


Figure 5.10: A parallel control fragment with its Petri net

- *Parallel* (*PAR*): this control fragment consists of multiple interleaved segments (i.e., regions in a protocol or branches in an activity diagram). The protocol in Figure 5.10 (a) has a parallel control fragment with two regions. Each region has one message: the 'a' message and the 'b' message. These two messages can be sent in an unspecified order. Either the 'a' message first and the 'b' message second or the 'b' message first followed by the 'a' message. Similarly, the activity diagram in Figure 5.10 (b) starts with a *fork* that splits the execution flow into two interleaved branches, which indicates that the two branches can be executed in any order. Considering the two branches in Figure 5.10 (b), the 'a' step may be performed before 'b' or the other way around. These forked branches are then joined, through a *join* UML-control entity, with the main execution flow.

  The equivalent Petri net to the parallel control fragment, based on Poutakidis' transformation approach [Padgham et al. 2005b], is shown in Figure 5.10 (c). The Petri net implements an intermediate level through two states ('$S0'$ ' and '$S0'$ '). This intermediate level allows the two entities ('a' and 'b') to occur in any order. For example, if 'a' occured, then the Petri net will be in the '$S0'$ ' state, allowing 'b' to occur before terminating the execution. Conversely, if 'b' occurred first, then the 'T1' transition would be fired, leaving the Petri net at the '$S0'$ ' state to allow 'a' to occur before reaching the termination place '$S1$'.

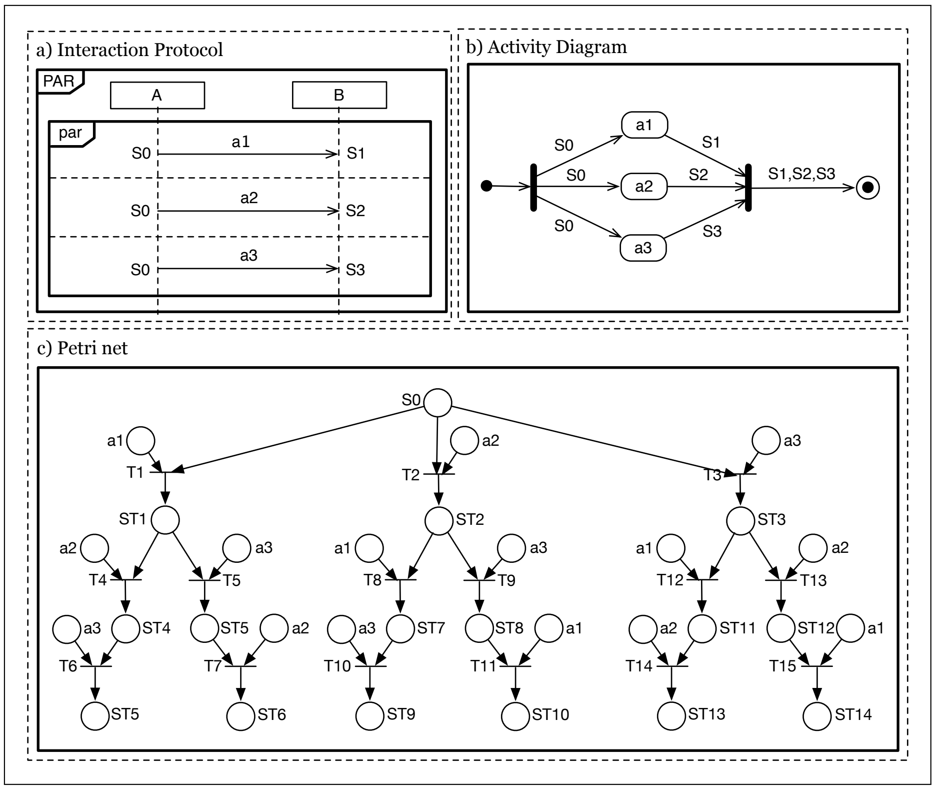


Figure 5.11: Original parallel control fragment transformation algorithm

5.1.2 Parallel Construct Translation

We directly adopt the translation approach outlined in Section 5.1.1, except for the translation of the parallel control fragment. Poutakidis' transformation of the parallel control fragment (as cited in [Padgham et al. 2005b]) is correct. However, the *complexity* of the *current algorithm is exponential*. For example, the Petri net in Figure 5.10 (c) models a parallel control fragment with two segments and each segment includes only one entity. Figure 5.11 (c) shows the Petri net version of the diagrams in Figures 5.11 (a) and 5.11 (b). As can be seen from the figures, the parallel control fragment has three segments, with only one entity in each segment. The additional segment, compared with the parallel control fragment in Figure 5.10, results in more levels of the Petri net (i.e., in a vertical increase). Further, the sequence of entities within each segment can lead to more places for each level (i.e., a horizontal increase).

Poutakidis' approach (as cited in [Padgham et al. 2005b]) can be problematic in the context of our framework. In our method, particularly in the case of checking agent-behaviour models against requirements, activity diagrams are usually complex. This is because requirements rep-

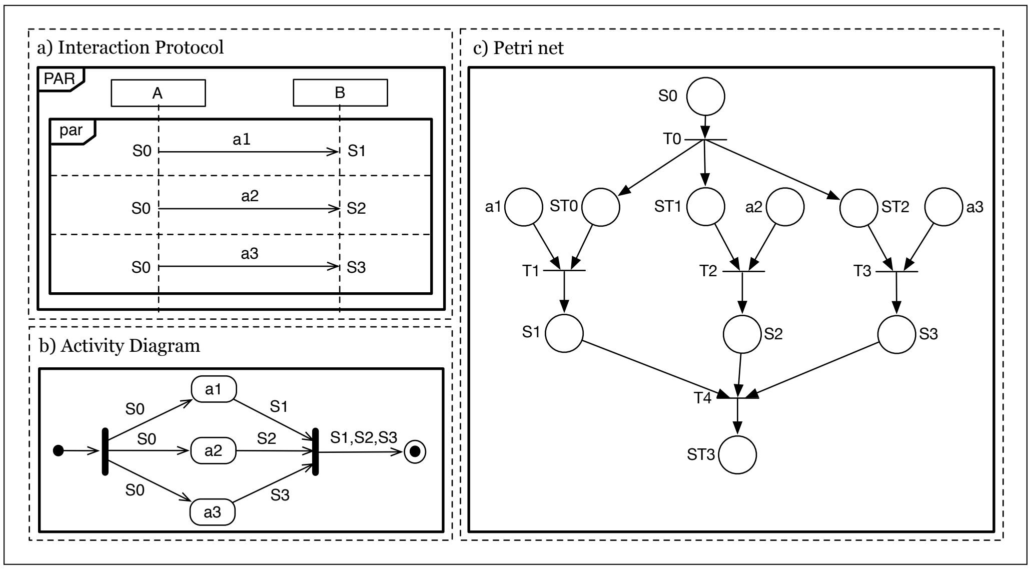
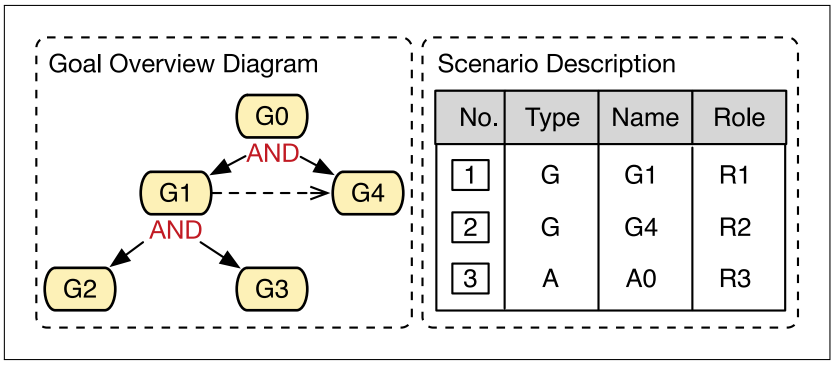


Figure 5.12: Modified parallel control fragment transformation algorithm

resent steps from scenarios along with relevant information for the goal model. Specifically, the parallel control fragment in these diagrams usually has more than two branches and it may include other control fragments (i.e., nested structures). Hence, if adopting Poutakidis' (as cited in [Padgham et al. 2005b]) transformation algorithm, the equivalent Petri nets to these diagrams would be very large and costly to create. As a result, we have simplified the transformation of the parallel control fragments so it suits the needs of our framework.

Instead of the *eager* approach used by Poutakidis (as cited in [Padgham et al. 2005b]), we use *lazy* approach to transform the parallel control fragment. We modify the transformation algorithm so it *forks* the interleaved branches via a *fork transition* and then *joins* them through a *join transition*. We create a separate Petri net for each segment in the parallel control fragment and then we connect these nets by a fork and a join transitions. Figure 5.12 (c) depicts the modified Petri net version of the parallel control fragment. As can be seen from the figure, each segment is modelled as a separate net and these three nets are linked through two transitions. The 'T0' transition forks the execution into three nets, while the 'T4' joins these three forked branches into one place (the 'ST3' place). Note that the Petri net in Figure 5.12 (c) is smaller in size compared with the one in Figure 5.11 (c).



No.	Type	Name	Role
1	G	G1	R1
2	G	G4	R2
3	A	A0	R3

Figure 5.13: Scenario and goal overview diagram

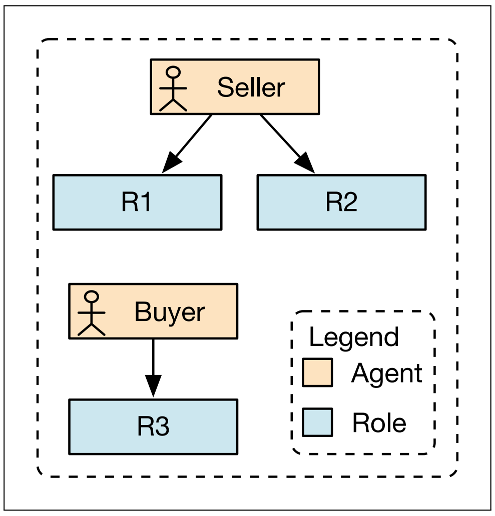


Figure 5.14: Part of the role grouping model relevant to the scenario in Figure 3.7

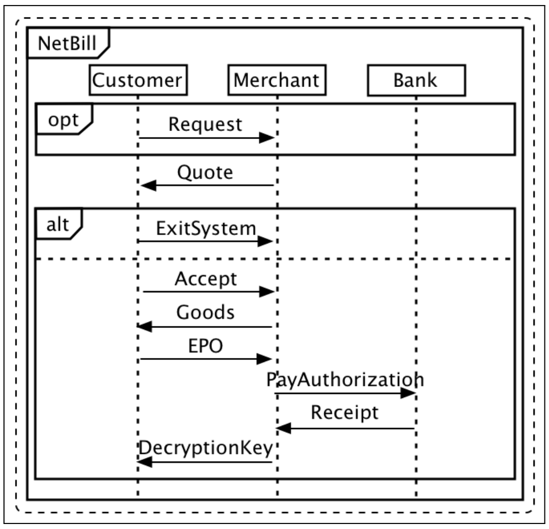


Figure 5.15: Net-Bill AUML protocol

5.2 Executable Behaviour Models

Agents are modelled so that they behave according to their corresponding point-of-reference artefacts (i.e., scenarios and interaction protocols). To verify these behaviour models against a given point-of-reference artefact, we extract all possible behaviour runs from the agent-behaviour models and check them against the point-of-reference artefact. Often, a realisation of a given point-of-reference artefact (a scenario or a protocol) is spread across multiple agents (i.e., in multiple agent-overview diagrams). The steps of a scenario are associated with roles and these roles can be assigned to multiple agents. For example, the scenario in Figure 5.13 has three steps, each of which corresponds to a different role. Figure 5.14 shows the role-grouping model of the system-to-be. The figure demonstrates that the roles 'R1' and 'R2' are linked to the 'Seller' agent, whereas

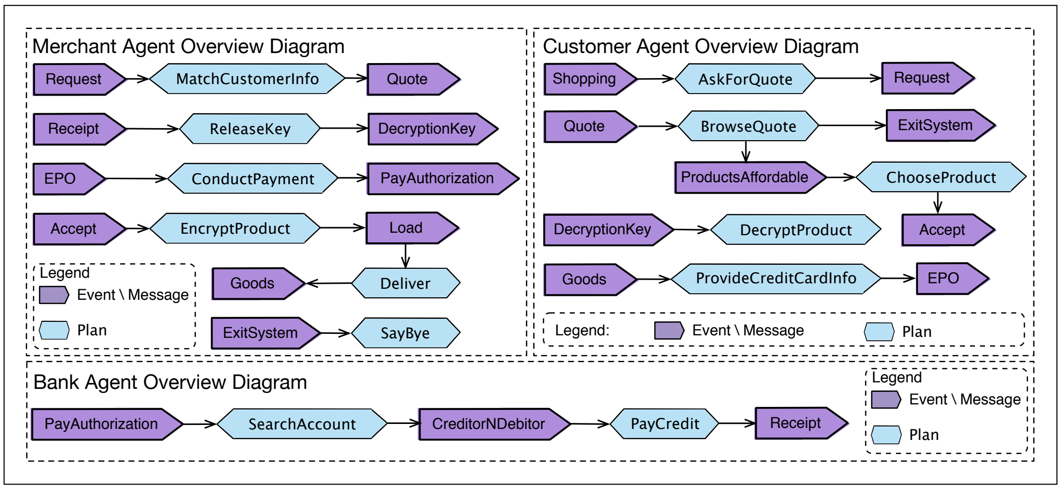


Figure 5.16: Agent-behaviour models for the Net-Bill AUML protocol

the role ‘R3’ is linked to the ‘Buyer’ agent. Therefore, the scenario in Figure 5.13 is realised through multiple agents (the ‘Seller’ and the ‘Buyer’).

Protocols often involve multiple agents, and thus, the realisations of such protocols are often spread across the agents that are involved in these protocols. For example, the ‘Net-Bill’ protocol[2] in Figure 5.15 models the interactions between three agents: ‘Customer’, ‘Merchant’ and ‘Bank’. The ‘Customer’ agent optionally sends a ‘Request’ message to the ‘Merchant’ agent at the start of the interaction. Then, the ‘Merchant’ agent responds by sending the ‘Quote’ message. After that, the ‘Customer’ agent may refuse or accept the quote. In the case where the customer accepts the quote by posting an ‘Accept’ message to the ‘Merchant’ agent, the delivery of the goods along with the payment transaction takes place.

Figure 5.16 represents one possible design—specified using the Prometheus methodology—that satisfies the protocol; the figure shows the *behaviour models* of the three agents, where the plans that handle incoming messages and produce outgoing messages are specified.

To generate the set of all possible behaviour runs for a design over more than one agent, all design entities (i.e., steps or communication constructs) in these agent-behaviour models that are relevant to a point-of-reference artefact need to be located and organised into a single coherent artefact. Thus, following Miller et al. [Miller et al. 2010], we construct a plan graph for representing the information relevant to a given point-of-reference artefact.

[2]This protocol models the transactions in electronic-commerce systems [Cox 1995].

The generation of the set of runs is a *two-step* process: *(1) the construction of the plan graph* (see Section 5.2.1); and, as explained in Section 5.2.3, *(2) the transformation of that plan graph to an executable model* so we can extract the behaviour runs.

5.2.1 Behaviour Designs to Plan Graphs

Miller et al. [Miller et al. 2010] propose an approach to check the coverage of interaction protocol test cases. They construct a *plan graph* for each protocol to define a set of coverage criteria for the test cases of each protocol. The plan graph aggregates the relevant messages and their plans from different participants into one structure. We have borrowed the concept of plan graphs to act as coherent structures that merge multiple designs. However, plan graphs of our approach are different from those proposed by Miller et al ([Miller et al. 2010]).

In our approach, we use plan graphs to allow for the extraction of all possible behaviour runs from the agent-behaviour models. Miller et al. [Miller et al. 2010] define a plan graph concerning an interaction protocol, so their plan graphs capture only the relationship between plans and messages. In contrast, we are interested in the relationship between plans and actions, percepts, messages, internal events and goals. This introduces many challenges as detailed in Section 5.2.1 (page 133).

Plan graphs in our approach are structures that show holistic view of agents with respect to the specified point-of-reference artefact. Each path through a plan graph represents one run of the system and each of these runs should conform to the requirements specified by the point-of-reference artefact under investigation. Formally, we define a *plan graph* as per Definition 1.

Definition 1. A *plan graph* is a directed bipartite graph $PG = (P, T, D)$, in which the two types of nodes are plans (P) and events (T) and the relationships between plans and events are represented by edges (D) where:

- P: is a set of plans $P = \{p_1, p_2, \ldots, p_n\}$.
- T: is a set of events $T = \{e_1, e_2, \ldots, e_n\}$.
- D: is a set of ordered pairs $D = \{(p_1, e_1), (e1, p_2), \ldots, (p_n, e_n), (e_n, p_{n+1})\}$.

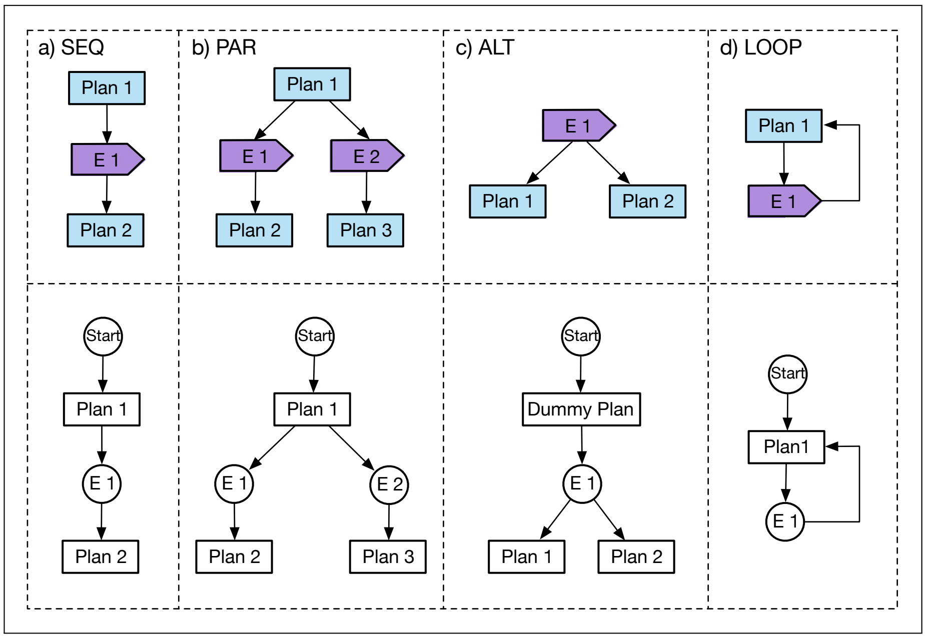


Figure 5.17: Control fragments that may be captured by a plan graph

Plan Graph Representation

Because of the links between events and plans (edges D in Definition 1), plan graphs can capture four different control fragments as follows (see Figure 5.17):

(1). Sequential (SEQ): $SEQ(p_1, \ldots, p_n)$ denotes that the plans within this fragment are to be executed sequentially.

(2). Alternative (ALT): $ALT(p_1, \ldots, p_n)$ denotes that only one of the plans within this fragment is to be executed.

(3). Parallel (PAR): $PAR(p_1, \ldots, p_n)$ denotes that the plans within this fragment are to be executed in parallel.

(4). Loop (cycles): cycles are specified by having a plan that handles an event post another event that is also handled by the plan that handles the first event ($p_1 \rightarrow t_2 \rightarrow p_1$).

Figure 5.18 shows the plan graph corresponding to the design of the 'Net-Bill' protocol (see Figure 5.15). This plan graph effectively merges the designs of the three agents in Figure 5.16, but includes information that is relevant to the specified protocol. It is important to note that plan graphs may include entities that are irrelevant to the specified point-of-reference artefact.

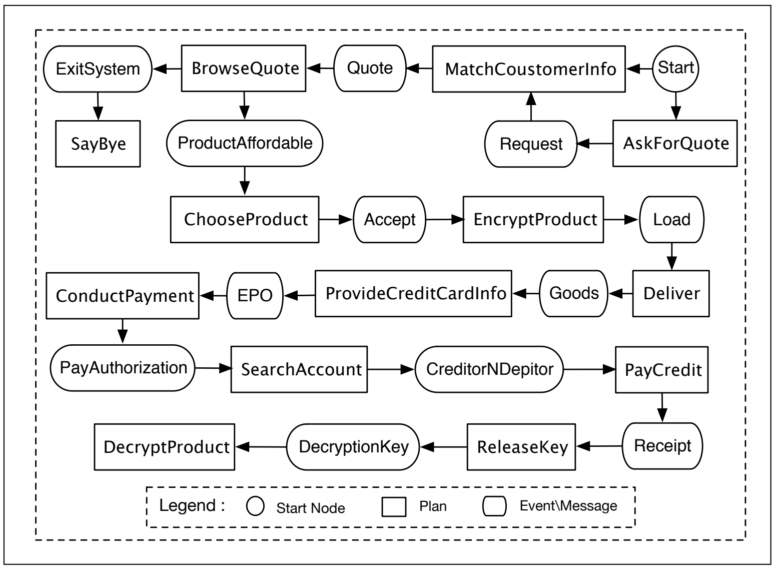


Figure 5.18: Plan graph for the Net-Bill design from Figure 5.16

For example, the ‘Load’ event in the plan graph is irrelevant to the protocol. The plan graph in Figure 5.18 captures three control fragments: alternative, parallel and sequential. The ‘Start’ node is linked to two plans forming an alternative control fragment. The ‘MatchCoustomerInfo’ plan and the ‘BrowseQuote’ plan must be executed in sequence. Since the ‘BrowseQuote’ plan posts two entities (‘ExitSystem’ and ‘ProductAffordable’), it implements a parallel control fragment, as all the entities interleave together.

Plan Graph Construction

The starting point for constructing a plan graph is the agent-behaviour models that consist of a collection of plans. Each plan has a defined trigger, which can be a goal (i.e., an internal event), a message or a percept. Plans can trigger sub-goals, send messages and perform actions. We create a plan graph from the detailed design by beginning with the design entity that corresponds to the first entity of the specified point-of-reference artefact and then by recursively traversing links in the agent behaviour models (see Algorithm 1). These links may be spread over several agents; for example, via the sending of a message. Therefore, the extracted plan graph is a subset of the agent (merged) behaviour models. For example, to generate the plan graph in Figure 5.18, the construction process starts with the plan that posts the first message in the protocol (either the ‘Request’ message or the ‘Quote’ message). Then, it follows the links between the plans

Algorithm 1 The algorithm for constructing a plan graph

```
Require: PoR (point-of-reference artefact: scenario or protocol.)
Require: DD (Agent detailed designs, i.e., agents' behaviour models).
 1: Initialisations:
 2: $node \leftarrow null$        # object of type Node with two attributes: (1) name and (2) list of type Node
 3: $firstEntity \leftarrow PoR.getFirstEntity()$        # fetches the first entity out of the PoR.
 4: $planGraph \leftarrow Null$        # an object of type node that stores the starting node of the plan graph.
 5: $node.name \leftarrow DD.getNode(firstEntity).getName$  # obtains the node name from the agents' designs.
 6: Begins:
 7: $planGraph \leftarrow ConstructPlanGraph($node.name$)$
 8: Function ConstructPlanGraph
 9: Pass In: nodeName:String
10: Pass Out: node:Node
11: if DD.getNode(nodeName).getLinkedNode = null then
12:     return node
13: else
14:     $node.name \leftarrow DD.getNode(nodeName).getLinkedNodeName$
15:     $node.list \leftarrow + DD.getNode(nodeName).getLinkedNode$
16:     ConstructPlanGraph ($node.name$)        # Recursively call the function
17: end if
18: EndFunction
```

throughout the behaviour models of the three agents.

There are five specific points that need to be addressed and that require a deviation from a simple subset of the agent designs (i.e., [Miller et al. 2010]).

First, there may be multiple entities that correspond to the first entity of the specified point-of-reference artefact. It is possible for protocols to have more than one initial message. For example, the protocol in Figure 5.15 has two possible starting points, as it begins with an optional control fragment. As a result, either the 'Request' message or the 'Quote' message is to be sent first. To overcome this issue, we add a single dummy 'Start' message as the first node of the plan graph, which is linked to all nodes that represent the protocol's starting points. As can be seen from Figure 5.18, the plan graph has its dummy 'Start' node linked to the two plans that can initiate the protocol.

With respect to scenarios, if the first step in the specified scenario is an abstract goal, then a subset of the goal's descendants could commence the scenario, meaning there may be a number of possible starting points for the scenario:

1. *OR-composed children:* In this case, the starting point is the goal or *one of* its children (i.e., its sub-goals). The plan graph is derived by following links from each of the possible starting points and then using these starting points as alternatives.

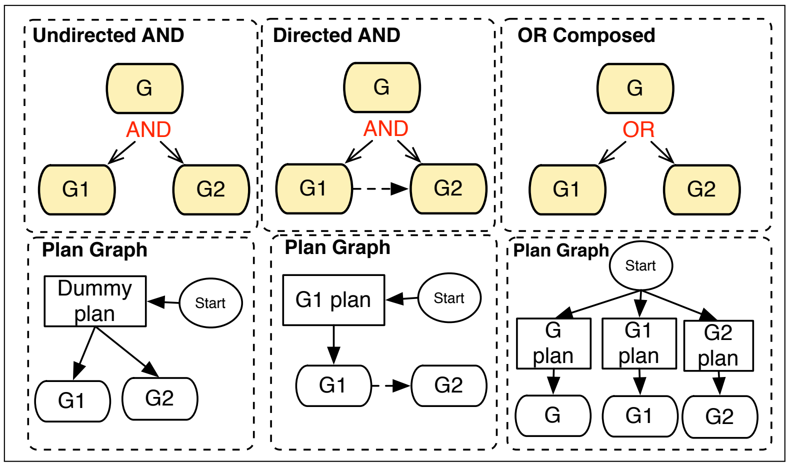


Figure 5.19: Plan graphs for the three types of goal compositions

2. *Undirected-AND children:* In this case, the starting point is any of the sub-goals, but all sub-goals must occur. The plan graph is derived by introducing a dummy plan that links to all of the sub-goals and that dummy plan is triggered by the start of the process.

3. *Directed-AND children*: The first sub-goal in the sequence is the starting point for the plan graph.

Figure 5.19 presents examples of each of these relationships and their corresponding plan graphs, where *G* is the (abstract) goal that is the first step of the scenario being considered.

After we have dealt with beginnings, we must deal with actions. In agent-behaviour models, an action is not followed by another step (it does not have any outgoing arcs) and, hence, it terminates the plan graph. However, in point-of-reference artefacts, an action does not always end the process. Thus, an *action* entity in the specified point-of-reference artefact creates a gap in the plan graph relative to the point-of-reference artefact. For example, in the *'RefundTransaction'* protocol in Figure 2.7 (page 26), the *'ReleaseCash'* action creates a gap in its detailed design (see Figure 2.9, page 28). This is because the action node is not linked to any entity in the design (i.e., it does not terminate the flow of the links in the design); however, the action does terminate the protocol. Similarly, the *'EnteredCredentials'* action in the *'LostCard'* scenario (see Figure 2.5, page 25) causes a gap in the detailed design of the agents, though this action does not indicate the end of the scenario.

We overcome this issue through segmenting the point-of-reference artefact and then using

Algorithm 2 The algorithm for segmenting the specified point-of-reference artefact

```
Require: PoR (Scenario description or AUML textual notation for a protocol) #array of entities.
 1: Initialisations:
 2: $numberOfEntities \leftarrow PoR.getNumberOfEntities()$
 3: $entity \leftarrow null$
 4: $listOfSegments \leftarrow null$
 5: $tempSegment \leftarrow null$
 6: $entityPointer \leftarrow 0$
 7: Begins:
 8: while $numberOfEntities \neq 0$ do
 9:     $entity \leftarrow PoR.Entity[entityPointer]$          # get one entity at a time.
10:     $tempSegment \leftarrow entity$
11:     $entityPointer = entityPointer + 1$          # move the array pointer one step forward.
12:     $numberOfEntities = numberOfEntities - 1$
13:     if $entity.type = `Action'$ then
14:         $listOfSegments \leftarrow tempSegment$
15:         $tempSegment \leftarrow null$
16:     end if
17: end while
```

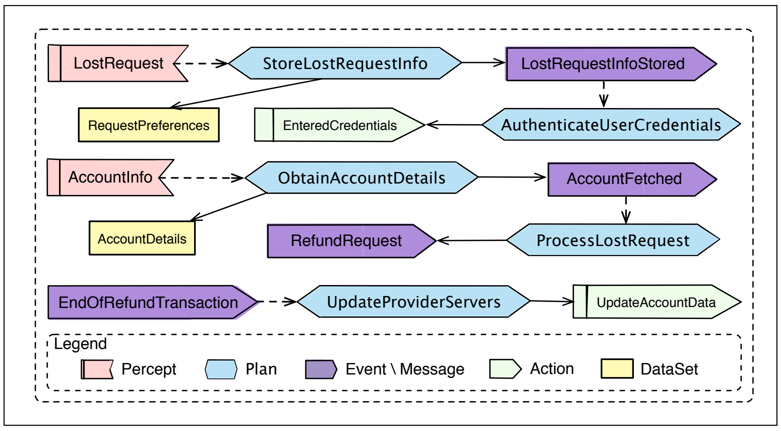


Figure 5.20: Ticketing Agent design overview diagram

these segments to *bridge the gaps*. Before construing the plan graph, we pre-process the specified point-of-reference artefact. This pre-process operation results in the segmentation of the specified point-of-reference artefact into multiple pieces based on the action entities. For example, the ‘LostCard’ scenario in Figure 2.5 (page 25) results into two segments. The first segment includes the first step (the ‘LostRequest’ percept) and everything up to the ‘EnteredCredentials’ action step, whereas the second segment starts from the ‘AccountInfo’ percept and continues until the ‘UpdateAccountData’ action step. After pre-processing the scenario, a plan graph for each segment is constructed.

Figures 5.20 and 5.21 show a possible design for the ‘LostCard’ scenario on page 25. The

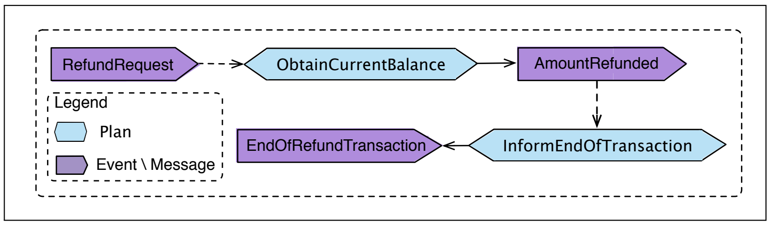


Figure 5.21: Payment Manager design overview diagram

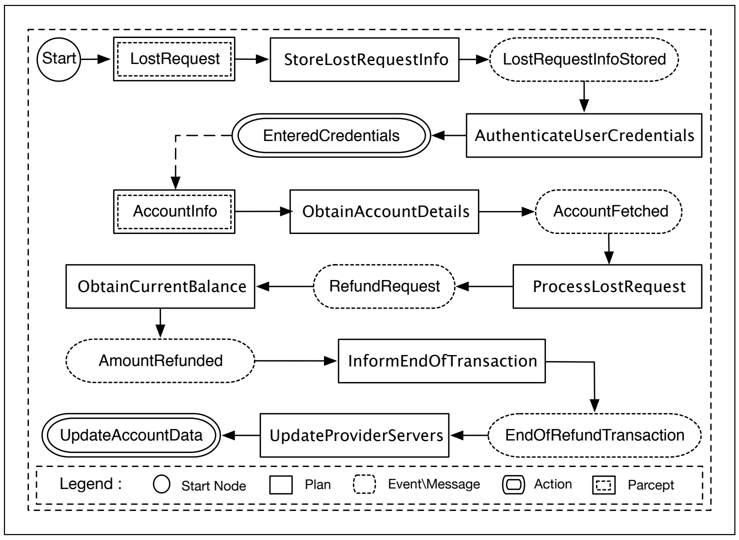


Figure 5.22: Plan graph of the scenario without goal nodes

plan graph that is created based on the scenario above captures two sub-graphs, one for each segment in the scenario (see Figure 5.22). The first sub-graph begins with the 'LostRequest' percept and ends with the 'EnteredCredentials' action node. The second sub-graph starts with the 'AccountInfo' percept and continues until the 'UpdateAccountData' action.

As stated earlier, we have rectified the gap issue by using the specified point-of-reference artefact to bridge the gap. If an action is not the final step of the point-of-reference artefact, then we use the entities defined in the point-of-reference artefact to determine the link to continue the flow of the plan graph. For example, if the point-of-reference artefact specifies that an entity following an action is a percept, then we use the percept as a continuation of the plan graph; if it is a goal, then we use the goal. We add a dashed link from the action to the corresponding continuation point. Considering the plan graph in Figure 5.22, we bridge the gap by linking—shown via a dashed arrow—the 'EnteredCredentials' action node that terminates the first segment with the 'AccountInfo' percept node that initiates the second segment in the 'LostCard' scenario.

Plan Name	**Goal/s to be realised**
StoreLostRequestInfo	ExtractRequePG-reductionstInfo
AuthenticateUserCredentials	VerifyCredentials
ObtainAccountDetails	GetAccouintInfo
InformEndOfTransaction	RefundOldCard
UpdateProviderServers	UpdateServersWithLostRequest

Table 5.1: Goals to be realised based on the descriptors of the plans in Figure 5.20 and Figure 5.21

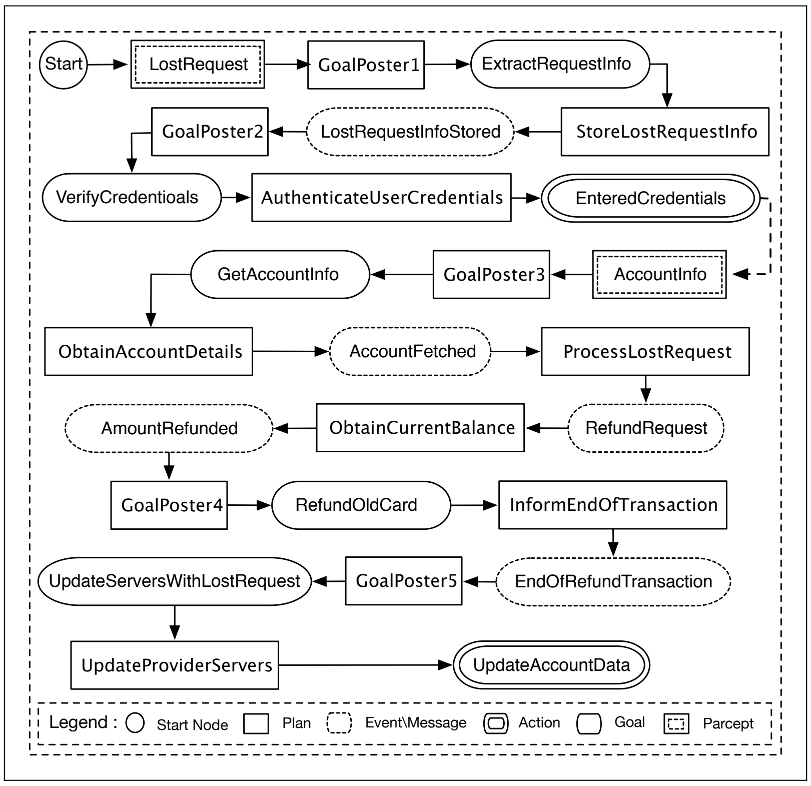


Figure 5.23: The plan graph in Figure 5.22 after incorporating the goals in Table 5.1

Third, Prometheus does not allow for the explicit modelling of goal steps in agent-behaviour models (i.e., for agents' detailed design overview diagrams). Instead, agent-behaviour models can have plans to realise these goal steps without the goal nodes appearing explicitly in the agent-behaviour models. Table 5.1 lists the goals corresponding to each plan shown in Figures 5.20 and 5.21.

We use plan descriptors for extracting the goals from the agent-behaviour models. The approach for constructing the plan graph is the same as explained earlier in this section. However,

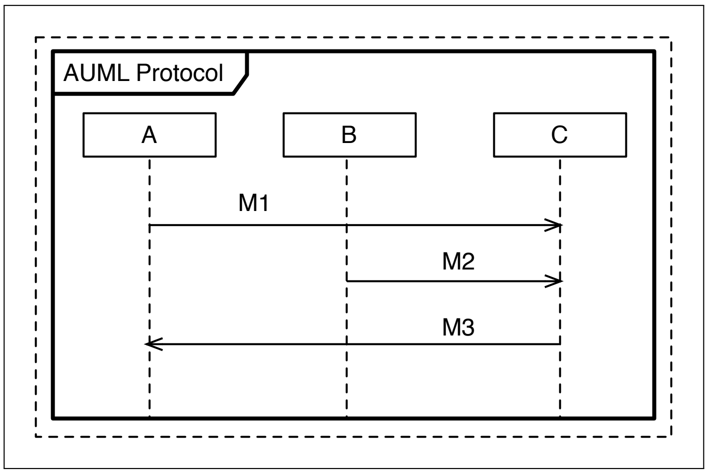


Figure 5.24: Simple AUML protocol with dependent messages

we insert the goals, which are extracted from the descriptors of the plans in the plan graph. We create a dummy plan to post the goal and tailor the plan that realises the goal step to handle it. After inserting these goals into the plan graph in Figure 5.22, we end up with a plan graph with all the relevant goals incorporated in it, as shown in Figure 5.23.

In the cases where the specified point-of-reference artefact is a scenario, it is possible for the first step to be assigned to a role that is associated with multiple agents, meaning that multiple plan graphs need to be considered. This is because multiple agents would realise the step. We consider the detailed designs of all agents that are linked to that role, procuring multiple plan graphs and positioning each plan graph as an alternative. Thus, the starting nodes of each plan graph will be children of a single starting node, forming one large plan graph from all designs.

Fifth, Prometheus allows plans to handle multiple events, with only one event as a trigger, meaning that the trigger occurs first. However, the other inputs and the output entities of such plans can occur in an unspecified order. Our approach must address this situation, in particular when there is a dependency between the entities of the specified point-of-reference artefact.

For example, the protocol in Figure 5.24 models the interaction between three agents: the ‘A’ agent, the ‘B’ agent and the ‘C’ agent. The flow of messages between these agents is as follows: ‘M1’ is sent first by the ‘A’ agent and then ‘M2’ by the ‘B’ agent. After the ‘C’ agent receives both ‘M1’ and ‘M2’ in the same order, it responds by sending ‘M3’ to the ‘A’ agent. Given this, ‘M3’ depends on the delivery of ‘M1’ and ‘M2’ and, hence, the protocol in Figure 5.24 highlights a dependency between its messages.

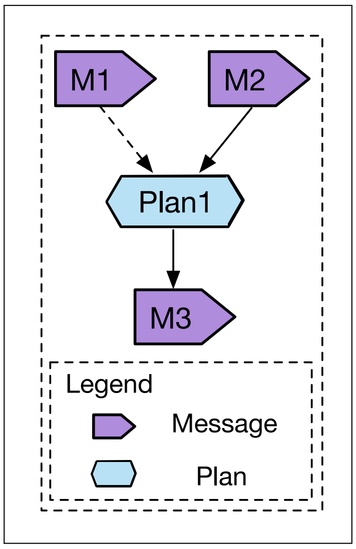


Figure 5.25: Agent 'C' design for the protocol in Figure 5.24

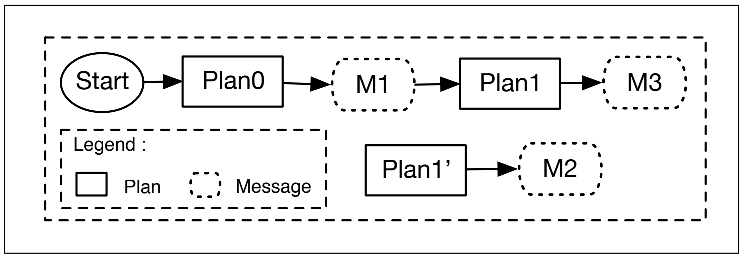


Figure 5.26: Plan graph based on the design of agent 'C' in Figure 5.25

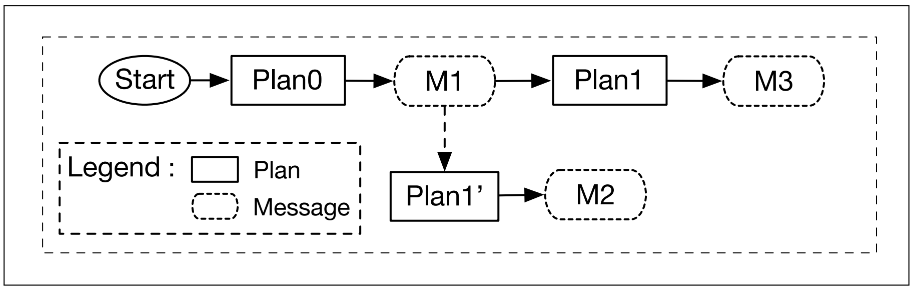


Figure 5.27: Using the protocol in Figure 5.24 to bridge the gap

Figure 5.25 depicts part of the agent-behaviour models with respect to the protocol in Figure 5.24, namely the 'C' agent. According to the design shown in the figure, the 'C' agent may send 'M3' after receiving'M1' or it may send it after processing 'M2'.

Figure 5.26 shows the plan graph constructed from the agent-behaviour models of the protocol in Figure 5.24. As per Figure 5.26, the plan graph captures a break in its flow ('Plan 1' forms another graph). Therefore, the plan graph in the figure has two separate threads, as shown in the

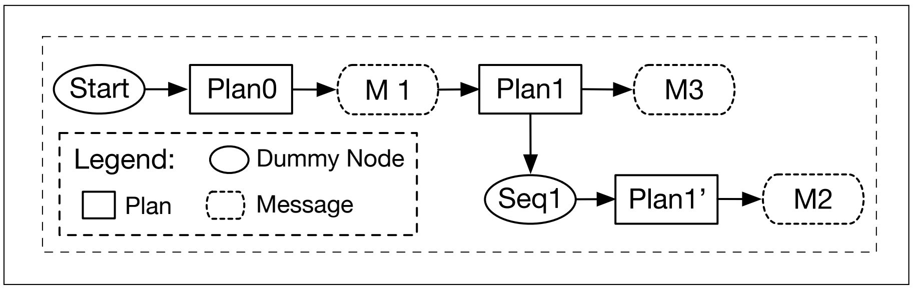


Figure 5.28: Bridging the plan graph through a dummy node

Figure 5.26. We cannot bridge this gap by linking these two threads via the sequence enforced by the protocol, as explained in the second challenge of this section. By doing so, we will end up with a plan graph that has two alternative paths, as shown in Figure 5.27. In fact, the plan graph in Figure 5.27 has different semantics to those stated by the agent-behaviour model in Figure 5.25.

Our approach bridges such breaks so that the semantics of the behaviour models are preserved. The bridging should ensure that the trigger occurs first, whereas the other inputs and output entities occur in an unspecified order. Therefore, we bridge such gaps in the flow via a dummy event ('Seq1' in Figure 5.28). This dummy event is posted by the same plan that handles the triggering event (see 'Plan1' in Figure 5.28). As a result, the input and the output entities ('M2' and 'M3') of the plan ('Plan1') are posted in parallel.

5.2.2 Plan Graph Reduction

A plan graph represents a subset of the agent-behaviour models that is relevant to a given point-of-reference artefact. It is not unusual for plan graphs to depict entities that are irrelevant to the specified point-of-reference artefact. This is because the construction algorithm used relies on the flow between entities across agent models and, consequently, all design entities within the flow are included. For example, the plan graph in Figure 5.23 has irrelevant entities to the 'LostCard' scenario (on page 25). The 'LostRequestInfoStored' event, the 'AccountFetched' event, the 'RefundRequest' message, the 'AmountRefunded' event and the 'EndOfRefundTransaction' message are not steps in the 'LostCard' scenario. Bearing in mind the purpose of plan graphs, which is to extract all possible behaviour runs from the agent-behaviour models concerning the specified point-of-reference artefact, all irrelevant tokens (e.g., the 'LostRequestInfoStored' event) can be filtered out from the runs.

Since a plan graph can capture parallelism, the number of possible behaviour runs can exponentially grow, particularly as the plan graph expands. This increase in the size of plan graphs may result from entities irrelevant to the specified point-of-reference artefact. Moreover, the parallelisms in such graphs can also be formed by irrelevant entities. Thus, reducing plan graphs by eliminating irrelevant entities will have a positive impact on the complexity of plan graphs and, consequently, will enhance the scalability of our approach.

In our work, we propose a *bottom-up* approach to reduce plan graphs. Our reduction approach involves the following two steps:

1. The elimination of unnecessary plans.
2. The elimination of unnecessary events.

Formally:

- Let Pr be the set of *events* (percepts, actions, goals and messages) in the specified point-of-reference artefact (i.e., scenario or protocol).
- As per Definition 1, a plan graph PG is (P, T, D).
- Let $T_{IR} = \{ t \mid (p, t) \in D \lor (t, p) \in D \} \setminus Pr$.
- Let $P_{IR} = \{ p \mid (p, t) \in D \land t \in T_{IR} \}$.
- Let $Ev = \{ t \mid (t, p) \in D \land p \in P_{IR} \}$

Rule 1. *Eliminate the unnecessary plans.* We *pre-process* the intended plan graph to eliminate the plans whose output set is irrelevant to the specified *point-of-reference* artefact. Formally:

$$D_{IR} = (D \setminus (\{ (t,p) \mid (t, p) \in D \land t \in T_{IR} \} \cup \{ (p,t) \mid (p, t) \in D \land t \in T_{IR} \}) \cup (Ev \times Pl))$$

Here:

- D_{IR} is the set of edges of the reduced plan graph.

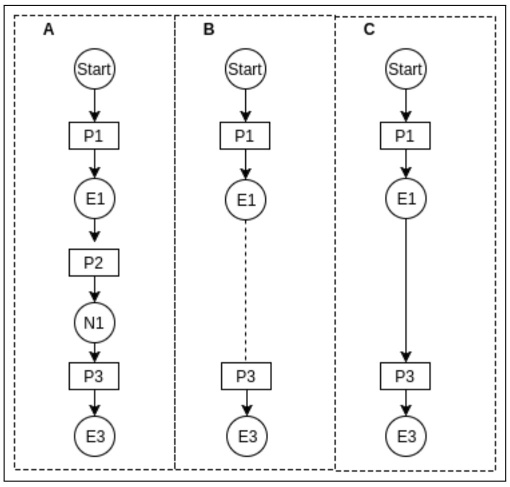
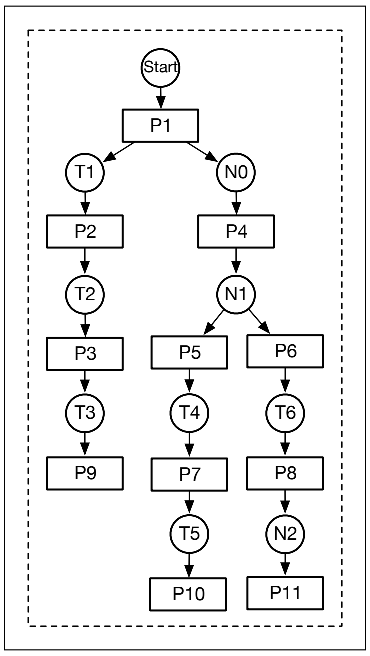
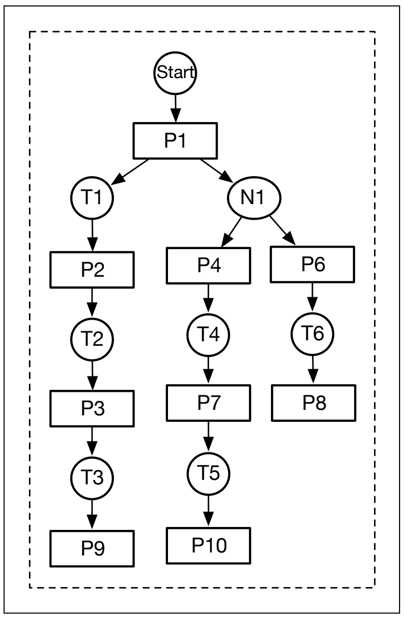


Figure 5.29: Example on the first reduction rule



(a) Origianl plan graph



(b) Reduced plan graph

Figure 5.30: Reducing a plan graph with ALT control fragment

It is important to note that this phase only eliminates the plans that have their output set of events irrelevant to the specified point-of-reference artefact and not the plans that handle these events. As a result, the new plan graph will have two segments. For example, let us assume that the 'N1' event in Figure 5.29 (A) is irrelevant to the specified point-of-reference artefact. Therefore, the plan 'P2' in the plan graph has its output set (i.e., 'N1') irrelevant to the specified point-of-reference artefact. According to Rule 1, this plan should be eliminated—a procedure that will divide the plan graph into two segments (see Figure 5.29 (B)). As shown in Figure 5.29 (C), the plan graph is bridged by linking the event 'E1' with the plan 'P3'.

After eliminating the unnecessary plans, we start examining the output and input sets of each plan from the bottom of the plan graph and commence removing the events that are irrelevant to the specified point-of-reference artefact (PR). It is important to note that if the irrelevant event is associated with an ALT control fragment (i.e., more than one plan to handle the same entity), then, to preserve the structure of the plan graph, the entity is not eliminated. Considering the plan graph in Figure 5.30a, let us assume that the 'N1' event is irrelevant to the specified point-of-reference artefact. Since 'N1' is associated with an ALT-control fragment, we cannot eliminate the event, as that would imply the removal of the plans that form the control fragment. Instead, we must keep the 'N1' event so that the structure of the original plan graph is preserved, as shown in Figure 5.30b.

Rule 2. *Eliminate the unnecessary events.* We eliminate the entities that are the irrelevant to the specified point-of-reference artefact and the plans that handle them. Formally:

$$D_{IR} = (D \setminus (\{(p,t) \mid (p,t) \in D \wedge t \in T_{IR}\} \cup \{(t,p) \mid (t,p) \in D \wedge t \in T_{IR} \wedge |\{S\}| = 1\} \cup (Pl \times Ev)))$$

Here:

- $S = \{(t,p) \mid (t,p) \in D \wedge t \in T_{IR}\}$
- D_{IR} is the set of edges on the reduced plan graph.

Unlike Rule 1, the Cartesian product in Rule 2 is $Pl \times Ev$. This is because we eliminate the plans that handle the irrelevant entities, whereas, in the first rule, we eliminate the plans that post them. Additionally, we constrain Rule 2 not to eliminate the events that are associated with more than one plan. This is achieved by restricting the cardinality of the set that contains pairs (t,p) to one ($|\{S\}| = 1$).

Considering the 'Net-Bill' protocol in Figure 5.15, the plan graph in Figure 5.18 captures three irrelevant messages: 'ProductAffordable', 'Load' and 'CreditorNDepitor'. Since the 'Load' and 'CreditorNDepitor' messages are the only outputs for their plans, they are eliminated as a

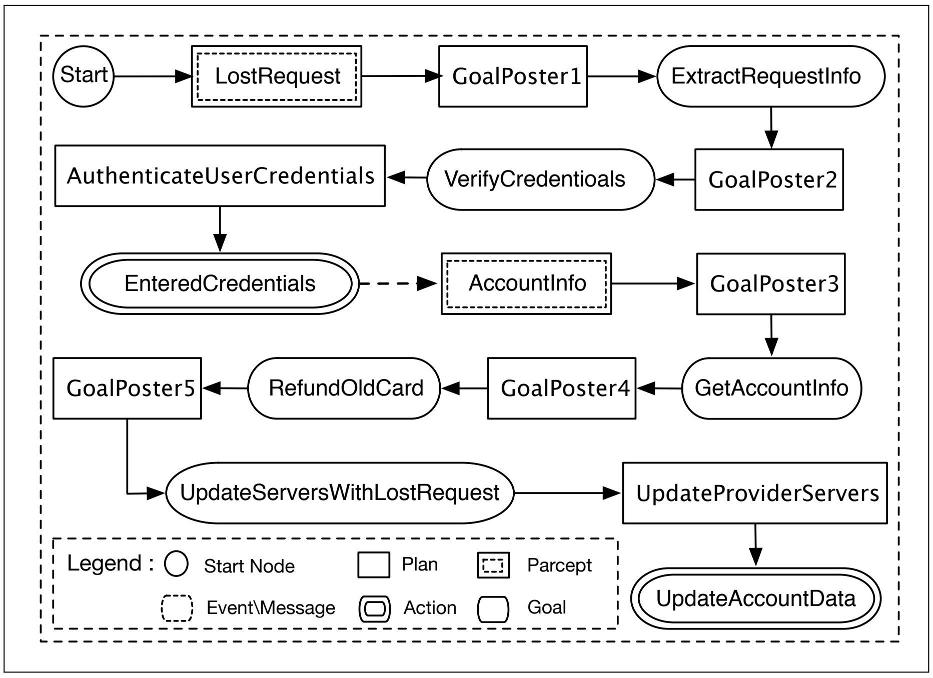


Figure 5.31: Reduced version of the plan graph in Figure 5.23

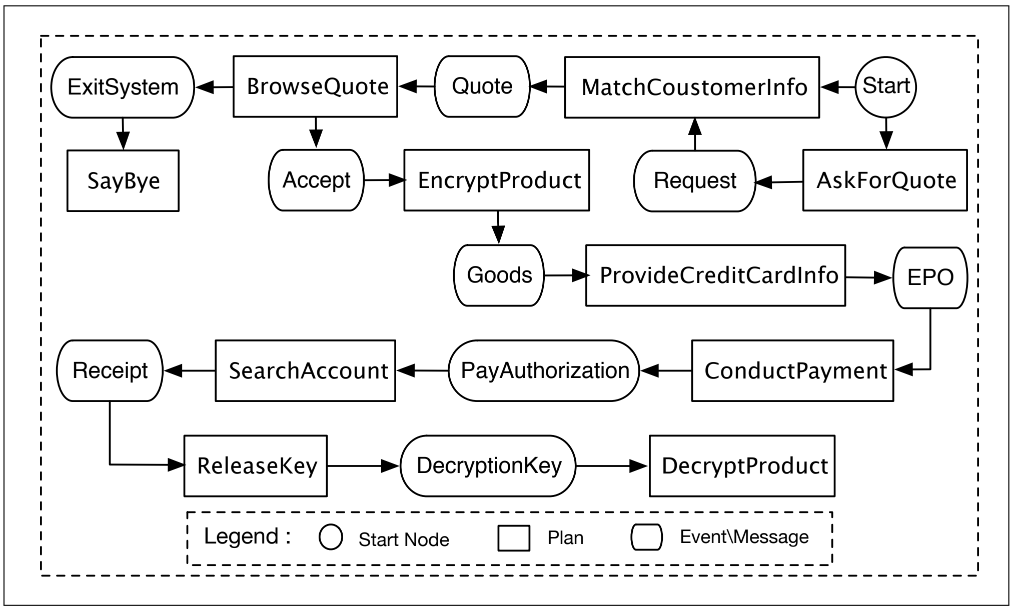


Figure 5.32: Reduced plan graph of the plan graph in Figure 5.18

consequence of removing their plans through the application of Rule 1. Based on Rule 1, all of these plans must be eliminated, as their output sets irrelevant to the protocol. Figure 5.31 shows the reduced version of the plan graph in Figure 5.23 after eliminating the unnecessary plans.

Conversely, the 'ProductAffordable' message in Figure 5.18 is not the only output for its plan

Run ID	Runs out of the PG	Runs / irrelevant entities	Unique runs out of the PG	Runs out of the RPG
1	$\langle 1,2,3,4,5,6,7,8,9,10,11,12\rangle$	$\langle 1,2,3,5,7,8,9,11,12\rangle$	$\langle 1,2,3,5,7,8,9,11,12\rangle$	$\langle 1,2,3,5,7,8,9,11,12\rangle$
2	$\langle 1,2,4,5,6,7,8,9,10,11,12,3\rangle$	$\langle 1,2,5,7,8,9,11,12,3\rangle$	$\langle 1,2,5,7,8,9,11,12,3\rangle$	$\langle 1,2,5,7,8,9,11,12,3\rangle$
3	$\langle 1,2,4,3,5,6,7,8,9,10,11,12\rangle$	$\langle 1,2,3,5,7,8,9,11,12\rangle$	$\langle 1,2,5,3,7,8,9,11,12\rangle$	$\langle 1,2,5,3,7,8,9,11,12\rangle$
4	$\langle 1,2,4,5,3,6,7,8,9,10,11,12\rangle$	$\langle 1,2,5,3,7,8,9,11,12\rangle$	$\langle 1,2,5,7,3,8,9,11,12\rangle$	$\langle 1,2,5,7,3,8,9,11,12\rangle$
5	$\langle 1,2,4,5,6,3,7,8,9,10,11,12\rangle$	$\langle 1,2,5,3,7,8,9,11,12\rangle$	$\langle 1,2,5,7,8,3,9,11,12\rangle$	$\langle 1,2,5,7,8,3,9,11,12\rangle$
6	$\langle 1,2,4,5,6,7,3,8,9,10,11,12\rangle$	$\langle 1,2,5,7,3,8,9,11,12\rangle$	$\langle 1,2,5,7,8,9,3,11,12\rangle$	$\langle 1,2,5,7,8,9,3,11,12\rangle$
7	$\langle 1,2,4,5,6,7,8,3,9,10,11,12\rangle$	$\langle 1,2,5,7,8,3,9,11,12\rangle$	$\langle 1,2,5,7,8,9,11,3,12\rangle$	$\langle 1,2,5,7,8,9,11,3,12\rangle$
8	$\langle 1,2,4,5,6,7,8,9,3,10,11,12\rangle$	$\langle 1,2,5,7,8,9,3,11,12\rangle$	$\langle 2,3,5,7,8,9,11,12\rangle$	$\langle 2,3,5,7,8,9,11,12\rangle$
9	$\langle 1,2,4,5,6,7,8,9,10,3,11,12\rangle$	$\langle 1,2,5,7,8,9,3,11,12\rangle$	$\langle 2,5,7,8,9,11,12,3\rangle$	$\langle 2,5,7,8,9,11,12,3\rangle$
10	$\langle 1,2,4,5,6,7,8,9,10,11,3,12\rangle$	$\langle 1,2,5,7,8,9,11,3,12\rangle$	$\langle 2,5,3,7,8,9,11,12\rangle$	$\langle 2,5,3,7,8,9,11,12\rangle$
11	$\langle 2,3,4,5,6,7,8,9,10,11,12\rangle$	$\langle 2,3,5,7,8,9,11,12\rangle$	$\langle 2,5,7,3,8,9,11,12\rangle$	$\langle 2,5,7,3,8,9,11,12\rangle$
12	$\langle 2,4,5,6,7,8,9,10,11,12,3\rangle$	$\langle 2,5,7,8,9,11,12,3\rangle$	$\langle 2,5,7,8,3,9,11,12\rangle$	$\langle 2,5,7,8,3,9,11,12\rangle$
13	$\langle 2,4,3,5,6,7,8,9,10,11,12\rangle$	$\langle 2,3,5,7,8,9,11,12\rangle$	$\langle 2,5,7,8,9,3,11,12\rangle$	$\langle 2,5,7,8,9,3,11,12\rangle$
14	$\langle 2,4,5,3,6,7,8,9,10,11,12\rangle$	$\langle 2,5,3,7,8,9,11,12\rangle$	$\langle 2,5,7,8,9,11,3,12\rangle$	$\langle 2,5,7,8,9,11,3,12\rangle$
15	$\langle 2,4,5,6,3,7,8,9,10,11,12\rangle$	$\langle 2,5,3,7,8,9,11,12\rangle$	-	-
16	$\langle 2,4,5,6,7,3,8,9,10,11,12\rangle$	$\langle 2,5,7,3,8,9,11,12\rangle$	-	-
17	$\langle 2,4,5,6,7,8,3,9,10,11,12\rangle$	$\langle 2,5,7,8,3,9,11,12\rangle$	-	-
18	$\langle 2,4,5,6,7,8,9,3,10,11,12\rangle$	$\langle 2,5,7,8,9,3,11,12\rangle$	-	-
19	$\langle 2,4,5,6,7,8,9,10,3,11,12\rangle$	$\langle 2,5,7,8,9,3,11,12\rangle$	-	-
20	$\langle 2,4,5,6,7,8,9,10,11,3,12\rangle$	$\langle 2,5,7,8,9,11,3,12\rangle$	-	-

Table 5.2: Behaviour runs out of the plan graph in Figure 5.18 and its reduced version in Figure 5.32 (refer to Table 5.3 for decoding the runs)

(the 'BrowseQuote' plan) and, hence, the plan cannot be eliminated. Instead, the 'ProductAffordable' message and its handler (the 'ChooseProduct' plan) are removed (according to Rule 2). Then, the event that is posted by the eliminated handler (the 'Accept' message) must be linked to the plan that posts the event that has been removed (the 'BrowseQuote' plan). Figure 5.32 depicts the reduced version of the plan graph in Figure 5.18. As per the figure, the original plan graph shown in Figure 5.23 is reduced in depth (from a depth of 13 to a depth of 8).

We have evaluated our plan graph reduction mechanism and the outcomes demonstrate that the original plan graph is equivalent to its reduced version. Both graphs result in the same set of behaviour runs after excluding the irrelevant entities from the runs in the original plan graph. For example, considering the original plan graph of the 'LostCard' scenario in Figure 5.18 and its reduced version in Figure 5.32, these graphs result in the set of behaviour runs listed in Table 5.2.

As per Table 5.2, the original plan graph results in 20 behaviour runs. The filtration of the irrelevant entities reveals in six repeated runs (e.g., run 14 and run 15 in Table 5.2). Thus, the original plan graph captures 14 unique behaviour runs and the reduced plan graph also captures 14 unique behaviour runs.

We evaluated the effectiveness of the reduction algorithm for 10 designs. Relatively ex-

Message Code	Message Name
1	Request
2	Quote
3	ExitSystem
4	ProductAffordable
5	Accept
6	Load
7	Goods
8	EPO
9	PayAuthorization
10	CreditorNDepitor
11	Receipt
12	DecryptionKey

Table 5.3: Coding for the messages in the Net-Bill protocol in Figure 5.16

	Original Plan Graph		Reduced Plan Graph		Depth Reduction	Parallelism Reduction
Design ID	Depth	#PAR	Depth	#PAR	Percentage (%)	Percentage (%)
1	26	7	17	5	34.6	28.5
2	10	1	8	0	20	100
3	20	4	14	1	30	75
4	15	1	8	0	46.6	100
5	14	2	7	0	50	100
6	24	8	17	4	29.2	50
7	12	2	9	1	25	50
8	11	2	7	1	36.4	50
9	19	1	8	1	57.9	0
10	17	9	16	7	6	22.2

Table 5.4: Plan graph vs. reduced plan graph

perienced developers developed these designs based on the Prometheus methodology. We also included an industry-level project: an air-traffic management system (Case 10). This project was designed by following the Prometheus methodology.

We generated the full plan graph for each design according to its corresponding point-of-reference artefact. We logged the depth of each plan graph and the number of parallel control fragments that each plan graph captured. Then, these plan graphs were reduced as per the reduction rules explained earlier in this section. Similarly, the depth of each reduced plan graph and the number of its parallelisms it has was also logged. We considered plans in counting the depth of both plan graphs and their reduced versions, as they are the executable entities in these graphs. With regard to the number of parallel control fragments, we counted the number of plans that posted more than one entity.

Table 5.4 lists all of the 10 cases considered in our evaluation. Our results show that the reduction algorithm is effective in reducing plan graphs. In the some cases, the algorithm reduces plan graphs to half (e.g., in cases 5 and 9). Further, the reduction in many incidents eliminated half, if not all, of the parallelism from the plan graphs (e.g., in cases 2 and 3). Reducing a plan graph means less runs to be extracted and, hence, less execution time. Thus, the reduction of plan graphs enhances the scalability of our approach, as will be discussed in Chapter 6 (page 162).

5.2.3 Plan Graph to Petri Net

Plan graphs are structures that provide static view of agents with respect to a particular point-of-reference artefact. Each path through a plan graph represents a possible sequence of plans and corresponding entities (i.e., steps or communication constructs) for the specified point-of-reference. Considering the plan graph in Figure 5.32, the sequence of plans—'Start, MatchCustomerInfo, Quote, BrowseQuote, ExitSystem, SayBye'—is part of a possible path for the plan graph and represents an instance of the dynamic behaviour of the system. Two sequences of messages result from evaluating the plans in the aforementioned sequence: 'Request, Quote, ExitSystem, Accept' and 'Request, Quote, Accept, ExitSystem'. These behaviour runs should conform to the requirements specified by the interaction protocol in Figure 5.15.

Because of the non-deterministic nature of BDI-agent frameworks[3], plan graphs must be traversed to extract all possible behaviour runs. In our method, we assume that parallelism represents the non-deterministic interleaving of both plans and other design entities, as this would accord with the observable behaviour demonstrated by most BDI frameworks. Consequently, a simple depth-first-traversal is not sufficient for extracting runs.

To extract behaviour runs, we must translate our plan graphs into marked Petri nets [Murata 1989]. We choose this formalism because Petri nets are expressive enough to model the semantics of our plan graphs and because our plan graphs are syntactically similar to Petri nets. Further, we can make use of existing theories for analysing the generated Petri nets.

[3]Plan choices are dependent on their context at the time of deliberation.

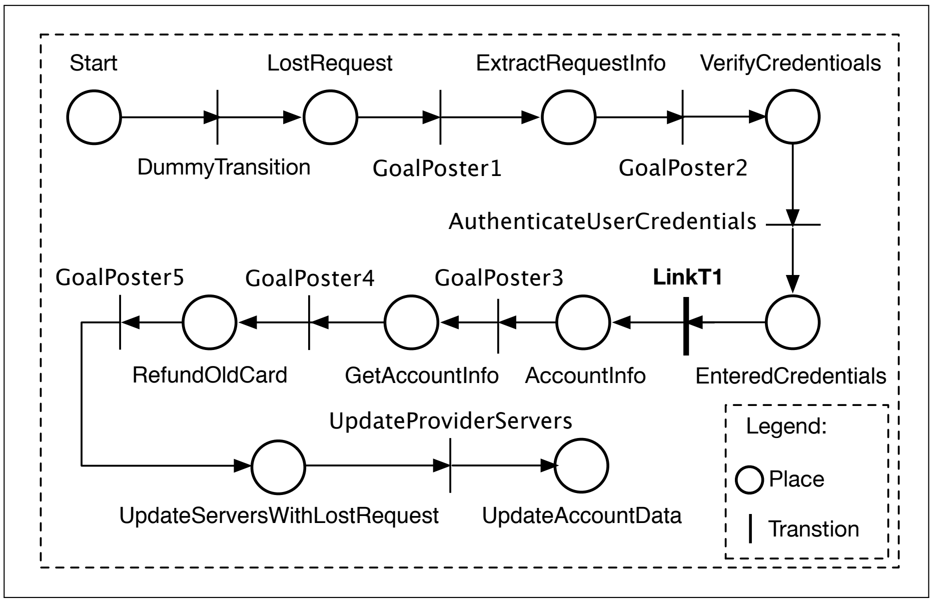


Figure 5.33: Petri net equivalent to the plan graph in Figure 5.31

Definition 2. A *marked Petri net* is a tuple $M = \{P, T, I, O, \mu\}$, in which P is a finite set of places, T is a finite set of transitions, such as $P \cap T = \emptyset$, I is an input function, O is an output function, and μ is the marking of the Petri Net defined as an n-vector, $\mu = \{\mu_1, \mu_2, \mu_3, ..., \mu_n\}$, in which $n = |P|$ and each $\mu_i \in N, i = 0, 1$.

Translating the plan graph into a Petri net is straightforward: plans are mapped to transitions, and other node types are mapped to places. Edges between nodes are mapped across. However, there are a few exceptions. First, *dashed edges*, which bridge the gaps in the plan graphs, are handled by creating a new transition node, which lies in between the two place nodes (e.g., 'LinkT1' in Figure 5.33).

Second, if the first node in the plan graph is a plan, then we create a start node (a place) that links to it. However, if the first node is not a plan, then we create a start node (a place) that connects to a new transition called StartT that links to the first node. As noted earlier, there can also be cases where there are a number of possible alternative starting nodes. In this case, we create—for each possible starting point that is not a plan—a corresponding transition and the start place node links to each of these transitions.

When a plan posts multiple entities (e.g., goals, actions or messages), the order is not specified in the design. In particular, with goals, actions and messages, the ordering of the steps to re-

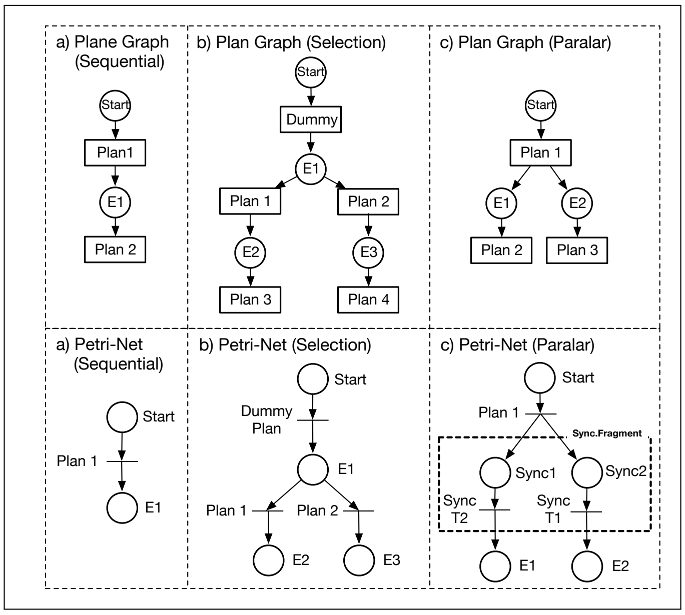


Figure 5.34: Petri nets' control fragments equivalent to plan graphs' control fragments

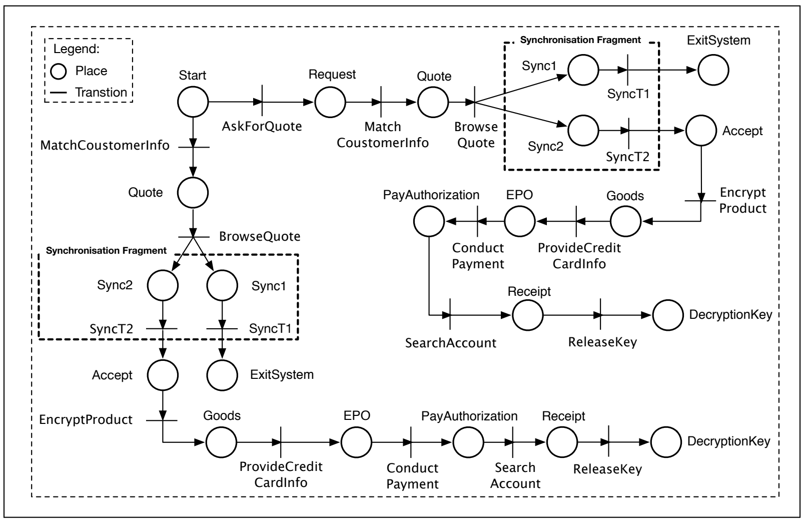


Figure 5.35: Petri net equivalent to the plan graph in Figure 5.31

alise these entities may be interleaved. Therefore, we treat the steps of a plan as though they were executed in parallel to allow for the different possible interleaving. That is, we create a separate asynchronous process for each step. The *synchronisation fragment* in Figure 5.34 (c) represents this process. Without this fragment, the steps will not be executed concurrently and, hence, all the behaviour runs will capture the same order between the respective steps. For instance, without the *synchronisation fragment* in Figure 5.35, a token would be simultaneously deposited on 'Accept' and 'ExitSystem', resulting in all runs with this order, whereas the plan graph does not specify such an order.

Finally, we exclude the plans that handle the last entities in the plan graph as they do not affect the run (both scenarios and protocols do not include plans) and would result in transitions without output places. Figure 5.35 shows the Petri net resulting from the translation of the plan graph of Figure 5.32.

Summary

In this chapter, we have presented an approach for transforming semi-formal, agent-based models to executable models. These models are then used to facilitate the automated correctness checking offered by our approach. To check the agent designs (i.e., the detailed structure of plans within agents), we compare all possible behaviour runs of the agent designs against the desired traces specified by the point-of-reference artefact. We transform the design models from Prometheus-specific, semi-formal models into Petri nets, which has two benefits. First, it generalises the approach, making it applicable to other methodologies; second, it allows us to leverage existing tools and techniques. Specifically, we transform the specified point-of-reference artefact (a scenario with a goal model or a protocol) into Petri nets and also translate agent-behaviour models into Petri nets. The Petri net equivalent of the agent design allows us to extract all possible behaviour runs. The Petri net of the point-of-reference artefact enables us to automatically check the conformity of the behaviour runs obtained from the agent-behaviour models against the specified point-of-reference artefact. We detail this checking process in the next chapter.

CHAPTER 6

Checking for Conformance

Our verification framework aims to check the conformity of all possible behaviour runs derived from the agent-behaviour models against the specified point-of-reference artefact. The previous chapters detailed the technicalities of producing verifiable components from the semi-formal agent models. Specifically, we explained the transformation of both the point-of-reference artefacts and the agent-behaviour models to Petri nets. The next step in our verification process is to extract all possible behaviour runs from the agent-behaviour models. Then, we will examine the validity of these runs against the Petri net of the specified point-of-reference artefact.

This checking process involves three phases:

(1) The extraction of the behaviour runs from the Petri net equivalent to the agent-behaviour models.

(2) The execution of these behaviour runs against the Petri net for the specified point-of-reference artefact.

(3) The reporting and logging of any discrepancies in the execution phase.

Figure 6.1 illustrates the modules of our verification framework with the *black/dark* parts representing the contributions of this chapter. Section 6.1 provides an explanation of the first two phases above (the ‘extraction and execution phases) and Section 6.2 details the reporting phase. Section 6.3 explains the coverage check that determines whether the agent-behaviour models cover

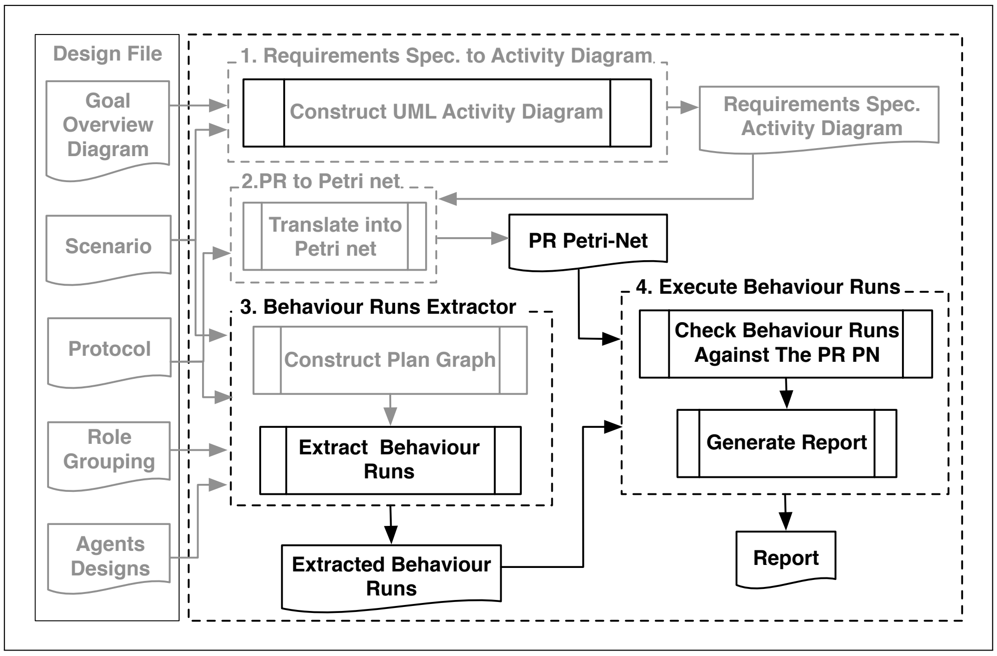


Figure 6.1: Verification framework (PR: point-of-reference artefact, PN: Petri net)

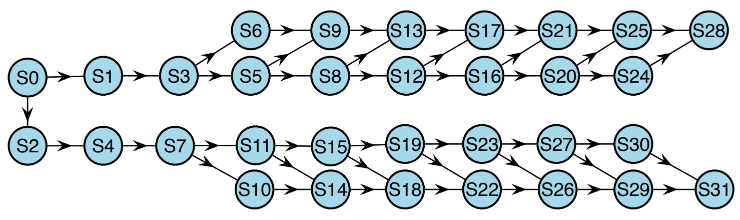
the specified point-of-reference artefact or not. We discuss the tool support for our verification framework in Section 6.4.

6.1 Conformity Check of The Agent-Behaviour runs

In Chapter 5, we showed how the parts of the agent-behaviour models that are relevant to the specified point-of-reference artefact are merged into a plan graph; then, the plan graph is transformed into a Petri net to extract its runs. In this section, we focus on the modules that extract and check the behaviour runs from the agent-behaviour models (Labelled 'Extract Behaviour Runs' and 'Check Behaviour Runs Against the PR PN' in Figure 6.1). The extraction module considers the Petri net that is equivalent to the plan graph as its input. Algorithm 3 details how both the extraction and the checks are performed through our approach.

Behaviour Runs Extraction

To obtain the behaviour runs, we calculate the reachability graph of the Petri net, which is a transition relation that defines the states and the transitions of the Petri net that is equivalent to the plan graph under consideration (see Section 2.6, page 37). Each path of the reachability graph



S0={1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S16={0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0}
S1={0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S17={0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
S2={0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0}	S18={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0}
S3={0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S19={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0}
S4={0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0}	S20={0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0}
S5={0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S21={0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0}
S6={0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S22={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0}
S7={0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0}	S23={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0}
S8={0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S24={0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0}
S9={0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S25={0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0}
S10={0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0}	S26={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0}
S11={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0}	S27={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0}
S12={0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S28={0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0}
S13={0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}	S29={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0}
S14={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0}	S30={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1}
S15={0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0}	S31={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1}

Marking Vector={Start, Quote, Sync1, Sync2, Accept, ExitSystem, Goods, EPO, PayAuthorisation, Receipt, DecryptionKey, Request, Quote, Sync1, Sync2, ExitSystem, Accept, Goods, EPO, PayAuthorisation, Receipt, DecryptionKey}

Figure 6.2: The reachability graph of the Net-Bill's Petri net in Figure 6.3

represents a possible execution of the Petri net from start to end. For example, the sequence of states $\langle\, S0,\ S1,\ S3,\ S5,\ S8,\ S12,\ S16,\ S20,\ S24,\ S28\,\rangle$ from Figure 6.2 represents a possible execution of the Petri net in Figure 6.3. Note that the Petri net in Figure 6.3 is equivalent to the plan graph in Figure 5.31 on page 145. When the reachability graph is in the state $S0$, this represents the marking

$\{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$.

Using the legend at the bottom of Figure 6.2, this means that a token is in the 'Start' place, because the first slot in the marking is one, while there are no other places with tokens. When the graph moves to the state $S1$, there is a token in the 'Quote' place (in the second column), indicating that the message 'Quote' in the 'Net-Bill' protocol in Figure 5.15 (page 129) has been sent. By analysing the changes in the markings between the transitions, we can infer which design entities are realised and in which order, as defined in the plan graph (see **line 20** of Algorithm 3).

Algorithm 3 The algorithm for checking the conformity of the possible behaviour runs out of the reachability graph

```
Require: $RG$ (Reachability graph)
Require: $S_0$ (Starting state in the reachability graph)
Require: $PoR$ (Scenario description or AUML textual notation for a protocol) #array of entities
Require: $legend$ (legend of the reachability graph)
 1: Initialisations:
 2: Set $ST$ to object of $RG$ states     # Stack of objects of type reachability graph states.
 3: Set $marking$ to null String         # A variable to store the marking vector corresponding to a state in reachability graph
 4: Set $run$ to null String             # A variable to store the tokens extracted from the states in reachability graph
 5: Set $token$ to null String           # A variable to store the token to be executed on the $PoR$ Petri net.
 6: Set $outcome$ to false               # This variable stores a boolean value of the execution outcomes.
 7: Set $s$ to state object              # This variable stores an object of type sate.
 8: Set $position$ to zero               # This variable stores the index of ‘1s’ within the sate’s marking vector.
 9: Set $runsCounter$ to zero
10: $ST.push(S_0)$
11: Begins:
12: while $ST$ is not empty do
13:     $s \leftarrow ST.pop()$
14:     if $run$ is not $empty$ then
15:         $run.truncate()$ #To remove the last token from the run with each pop operation so the run can be appended with the
            next token without re-traversing the reachability graph
16:     end if
17:     if $s.isVisited == false$ then
18:         $s.isVisited \leftarrow true$        # $isVisited$ is a mutator that allows storing a boolean value in it.
19:         $marking \leftarrow s.getStateMarking()$
20:         for each 1 in $marking$ do
21:             $position \leftarrow marking.getIndex()$ # $getIndex$ is a method that returns the indices of 1s within a
                marking vector.
22:             $token \leftarrow legend.getToken(position)$
23:             if $token$ is in the $PoR$ then
24:                 # The condition above is filtering out irrelevant tokens.
25:                 $run.add(token)$
26:                 $outcome \leftarrow executeToken(token, run)$ # Refer to Algorithm 4 for the $executeToken$
                    method.
27:                 if $outcome = false$ then
28:                     $reportingModule(s,\ run)$ # Refer to Algorithm 5 for the $reportingModule$ method.
29:                     $ST.POP()$        # This to eliminate the erroneous state and its sub-graphs.
30:                     $runsCounter++$
31:                     $break$
32:                 end if
33:             end if
34:         end for
35:         if $outcome = true$ then
36:             # The condition above is true if NONE of the ”executeToken” calls fail.
37:             if $RG.adjacentEdges(s)$=null then
38:                 $ST.POP()$            # Indicates the end of the current run.
39:                 $runsCounter++$
40:             end if
41:             for all edges in $RG.adjacentEdges(s)$ do
42:                 $ST.push(edges)$
43:             end for
44:         end if
45:     end if
46: end while
```

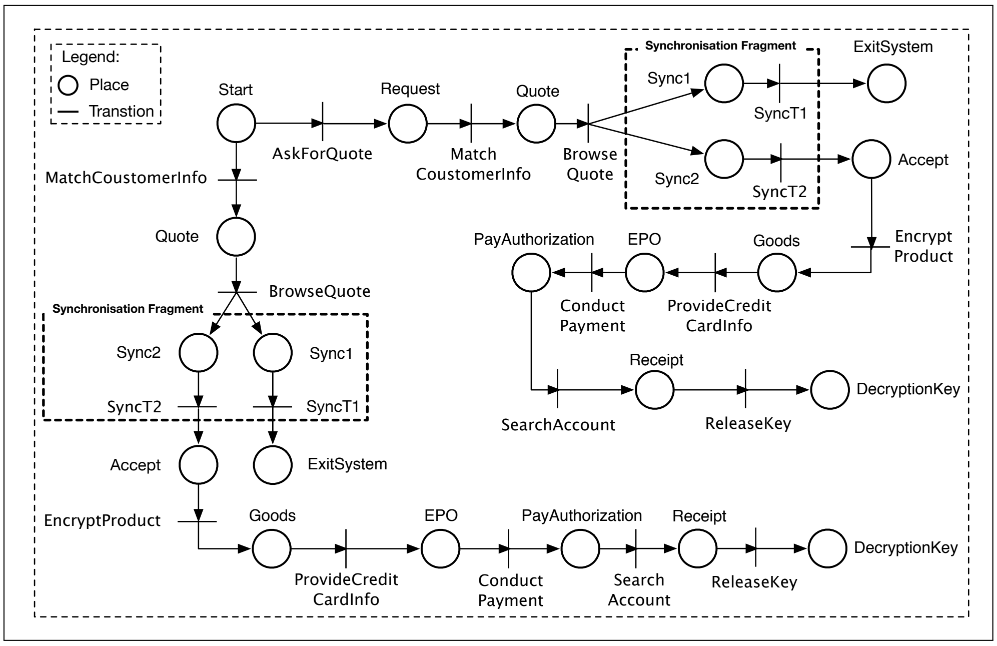


Figure 6.3: Petri net equivalent to the plan graph in Figure 5.31 on page 145

We use a standard depth-first traversal for the reachability graph (see the *loop* in **line 41**) to extract the set of all possible behaviour runs defined by the Petri net (and therefore its corresponding plan graph).

As a final step, we exclude any entities not related to the specified point-of-reference artefact. Specifically, we filter out of each execution run any elements that do not correspond to design entities (i.e., internal events, percepts, messages and actions) in the specified point-of-reference artefact (see Algorithm 3, **line 23**).

Considering the Petri net in Figure 6.3, the places added in the synchronisation fragment ('Sync1' and 'Sync2') are filtered from the behaviour runs. After mapping the marking vectors with the legend in Figure 6.2, the following behaviour run is generated:

$\langle\, Start, Request, Quote, Sync1, Sync2, ExitSystem, Accept, Goods, EPO, PayAuthorization,$
$Receipt, DecryptionKey\,\rangle.$

We take both the 'Sync1' and 'Sync2' tokens out of the run because they are irrelevant to the 'Net-Bill' protocol in Figure 5.15 (page 129). Hence, the behaviour run that needs to be checked for conformity against the point-of-reference artefact is as follows:

$\langle\, Start, Request, Quote, ExitSystem, Accept, Goods, EPO, PayAuthorization, Receipt,$
$DecryptionKey\,\rangle.$

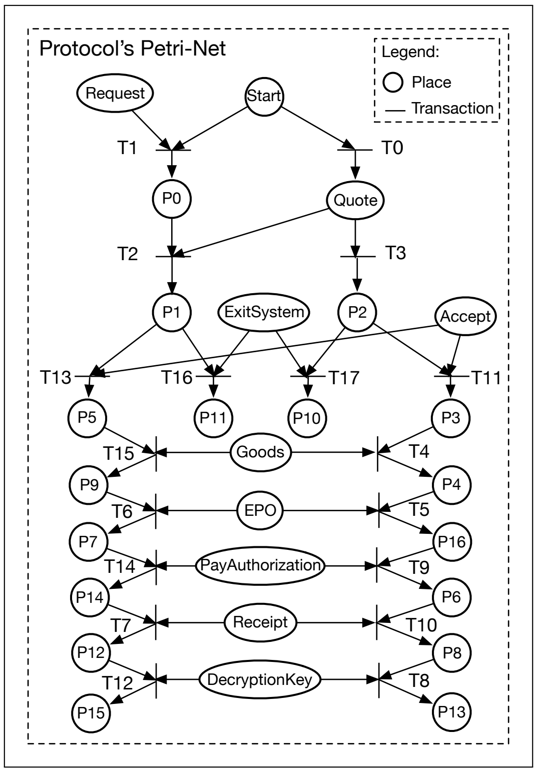


Figure 6.4: The Petri net of the Net-Bill protocol in Figure 5.15 (page 129)

Behaviour Runs Execution

Each possible behaviour run is verified by checking that it is a valid execution trace of the point-of-reference artefact Petri net (note: not the plan graph Petri net), using the entities of the behaviour run as tokens on the Petri net for the specified point-of-reference artefact.

Plan graphs can capture parallelism (see page 133). As a result, some designs can result in an explosive number of runs and, therefore, long execution times. To mitigate this problem, we have developed the depth-first search that extracts and executes one token from the reachability graph at a time. The method-call $executeToken(token, run)$ on **line 25** of Algorithm 3 implements such an approach. This method is detailed in Algorithm 4.

Figure 6.4 depicts the corresponding Petri net that models the semantics of the 'Net-Bill' protocol described in Figure 5.15 (page 129). Considering the Petri net of the 'Net-Bill' protocol in Figure 6.4, at the beginning, there is a token in the 'Start' place; the first message is taken from the marking of the current state of the reachability graph and a token is put into the Petri net place corresponding to that message. In the first iteration of the reachability graph in Figure 6.2,

Algorithm 4 The algorithm for executing the run's tokens

```
Require: Petri_net (a Petri Net representing the specified point-of-reference artefact)
 1: Initialisations:
 2: $transitionsStack \leftarrow$ null      # Stack of objects of type Petri net transitions.
 3: $previousRun \leftarrow null$      # Variable of type list to store the previous run from the calling method.
 4: $backTrackController \leftarrow zero$      # Counter to count the number backtracks.
 5: $outcome \leftarrow false$      # The return value of this method.
 6: $executedTransition \leftarrow null$      # This is a static variable to store the transition being executed.
 7: Function executeToken
 8: Pass In: token:String, run:List      # Method parameters.
 9: Pass Out: outcome:Boolean      # Method return value.
10: $previousRun \leftarrow run$
11: if run is a new run then
12:     $backTrackController \leftarrow run - previousRun$
13:     while $backTrackController \neq 0$ do
14:         $backTrackController = backTrackController - 1$
15:         $transitionStack.pop()$ #Implementing the execution backtracking.
16:     end while
17: end if
18: place token in Petri_net
19: $executedTransition \leftarrow Petri\_net.getTransition()$ #Get the transition to be executed based on the token.
20: if $Petri\_net.execute = fail$ then
21:     return $false$
22: else
23:     return $true$
24: end if
```

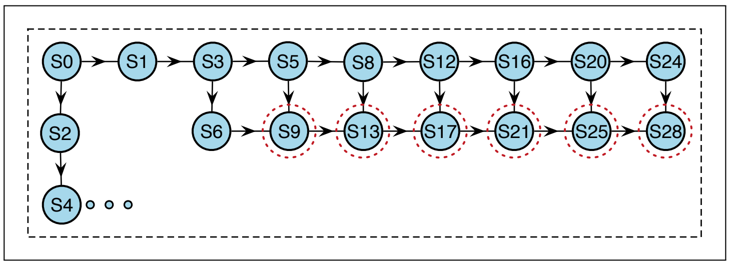


Figure 6.5: Part of the reachability graph in Figure 6.2

the 'Request' token is developed from the marking of the $S1$ state; then, a token is placed in its corresponding place in the Petri net, which is consequently executed. The transition 'T1' can fire because its input places contain tokens and the result is a token in the 'P0' place. The process then repeats for the next message in the run. In fact, in the second iteration, the run is appended with the next token from the reachability graph (the 'Quote' token out of the $S3$ state). Hence, the run is built gradually as its successful execution continues.

The gradual construction of behaviour runs allows for the determining of *erroneous states* in the reachability graph. As a result, we can eliminate the sub-graph underneath such states

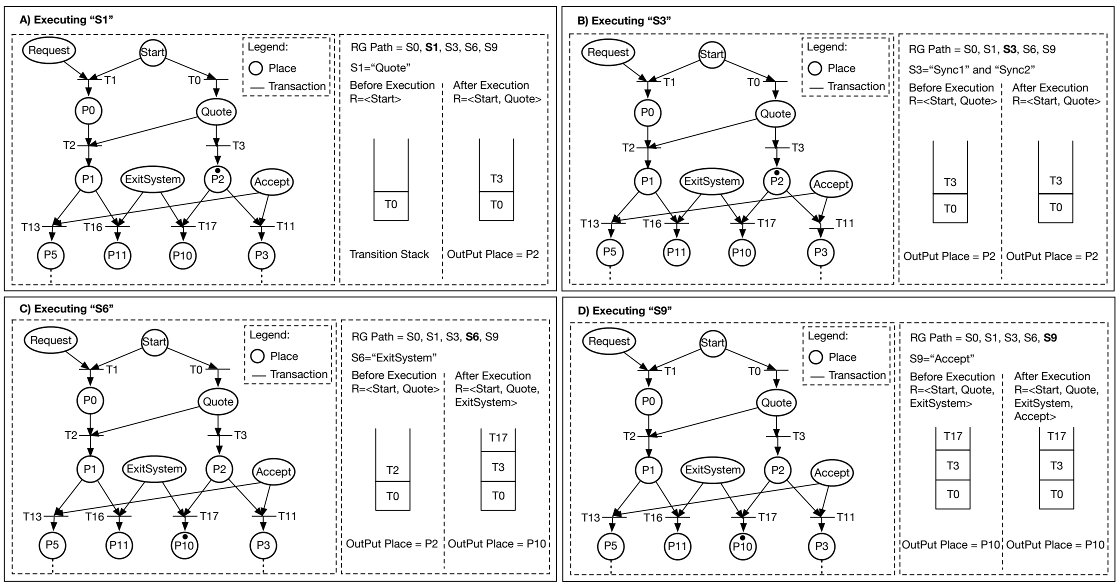


Figure 6.6: Petri net execution with backtracking

and save the traversal time. For example, the token from the $S9$ state ('Accept') in Figure 6.5 failed to execute in the Petri net in Figure 6.4. This allows for the marking of the $S9$ state as an erroneous state and, consequently, for the elimination of all the sub-graphs underneath the $S9$ state (see the dashed circles in Figure 6.5). However, in the worst-case scenario (where all runs pass successfully), this strategy does not save any time. Additionally, the progressive construction approach does not require the traversal to start from the beginning of the reachability graph each time. Therefore, the gradual construction of the behaviour runs reduces the cost of traversing the reachability graph.

The execution mechanism in our approach considers one token at a time, rather than executing the entire set of runs at once (see Algorithm 4). Therefore, the execution cost is reduced, as the execution does not start from the beginning of the Petri net each time a run is executed. The execution algorithm implements a *backtracking* mechanism. As a result, the execution of the Petri net starts from its branching point in the previous execution.

Backtracking is implemented via a stack of objects that keeps track of the execution information. As per **lines 1** and **3** in Algorithm 4, this process is managed through a stack that contains the visited transitions of the Petri net and an index (*backTrackController*) that tracks the run to be executed. Since the extraction trims and appends to the previous run, this information is used for controlling the execution. As per **line 11**, the difference between the length of the original run and the trimmed run is calculated to determine the number of the *pop* operations needed from the $transitionStack$. Finally, the length of the trimmed run is assigned to the $backTrackController$ variable.

Figure 6.6 illustrates the process of the conformity check for a path out of the reachability graph in Figure 6.2: $\langle\, S0,\, S1,\, S2,\, S3,\, S6 \ldots \rangle$. According to the traversal and execution mechanism of our approach, we extract and execute one token at a time (i.e., one state from the reachability graph is processed at a time).

In Figure 6.6 (a), before the $S1$ state, the behaviour run 'R' has only one token (the 'Start' token), and the transition stack has one transition ('T0'). After mapping the $S1$ state, the run includes the 'Quote' token and, hence, the 'T3' transition is fired and pushed into the transition stack.

As shown in Figure 6.6 (b), the status of both the run and the transition stack remain the same, as the 'Sync1'' and 'Sync2' tokens are irrelevant to the protocol and, hence, they are excluded from the run. Considering the $S6$ state in Figure 6.6 (c), the Petri net successfully fires the 'T17' transition, as both the 'P2' and the 'ExitSystem' places have tokens. However, the Petri net fails to execute concerning the $S9$ state, since it hits a termination place ('P10'), while the $S9$ state maps to the 'Accept' token (i.e., tries to execute the 'T11' or 'T13' transitions). Thus, the $S9$ state and its descendant states are marked as erroneous (see the dashed circles in Figure 6.5).

The next iteration extracts and executes a new path from the reachability graph. The extraction does not start from the beginning of the reachability graph. Instead, we reuse the previous path after truncating the irrelevant states and then we append the new states, forming another path. For example, the path that was checked in our earlier example is: $\langle\, S0,\, S1,\, S2,\, S3,\, S6,\, S9\,\rangle$. After marking $S9$ as an erroneous state, the graph traversal algorithm will backtrack to the $S3$ state, to where the reachability graph branches. Since the traversal backtracks two states back, the $S6$ and the $S9$ states are truncated from the path. The traversal then starts normally, by extracting the $S5$ state and appending it to the truncated path.

When executing the token corresponding to the $S5$ state, our execution mechanism calculates the difference in length between the previous path and the new path. In our example, the length of the last path is five, while that of the current path is four; thus, the difference in length is one. The

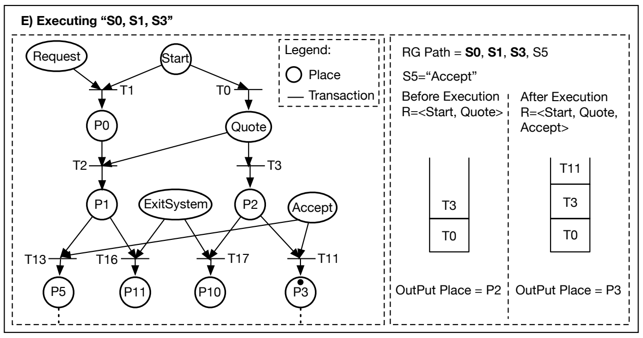


Figure 6.7: Petri net execution with backtracking

execution uses this difference to determine the number of pop operations required on the transition stack. By doing so, the execution algorithm implements the backtracking approach and does not start from the beginning of the Petri net each time. Figure 6.7 shows that the previous path is reused and appended by the $S5$ state and that the execution of the Petri net starts from the 'T3' transition.

Complexity Analysis

The complexity of our approach is *exponential* in the size of the resultant plan graphs. This is because the approach rests on the extraction of the possible behaviour runs from these plan graphs. The number of runs grows exponentially as the parallelism increases in the plan graph and the plan graph expands (i.e., there is an increase in the depth of the plan graph). We derive a formula that calculates how many runs a plan graph can have, based on the following parameters:

- A plan graph PG branches into b parallel branches (i.e., the breadth of the plan graph).
- d signifies the depth of the parallel branches. Note that we consider plans when determining the depth of the branches.

Given the definitions above, the number of possible ways of merging the branches in the plan graph PG is:

$$(b * d)!/((d)!)^{b}$$

	#Branches (b)	Depth (d)	#behaviour runs
Case 1	2	1	2
Case 2	2	2	6
Case 3	2	3	20
Case 4	2	4	70
Case 5	2	5	252
Case 6	3	1	6
Case 7	3	2	90
Case 8	3	3	1680
Case 9	3	4	34650
Case 10	3	5	756756
Case 11	4	1	24
Case 12	4	2	2520
Case 13	4	3	369600

Table 6.1: Exponential growth in number of runs as parallelisms and depth increase in plan graphs

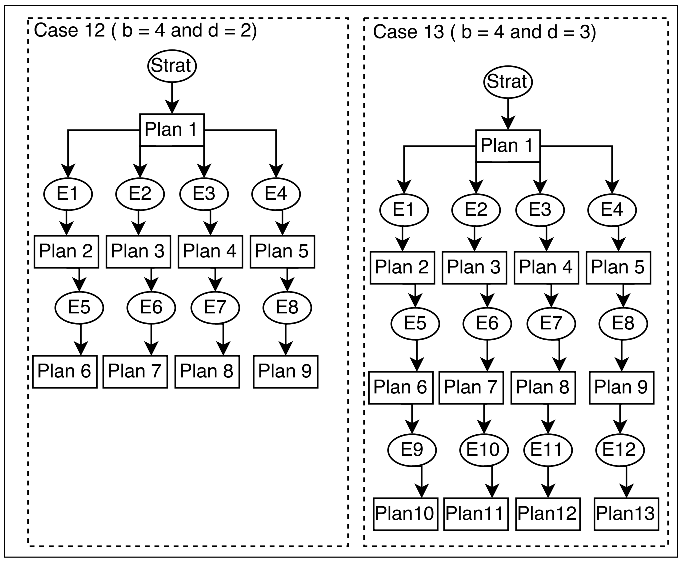


Figure 6.8: Examples on the exponential growth of behaviour runs as plan graphs grow in size

To obtain an idea of the number of runs that can be derived from a plan graph that follows the parameters above, a small study was undertaken. We synthesised a number of plan graphs that were structured according to the parameters described above. Using the tool-support of our verification framework (see Section 6.4), we recorded the number of all possible paths extracted from each plan graph. Then, we analytically calculated the number of paths.

Table 6.1 shows how the number of behaviour runs grows exponentially as plan graphs grow in size. Thus, the complexity is exponential in the branch factor and in the depth of the resulting plan graphs. A particular example of this is when the number of parallel branches increases and

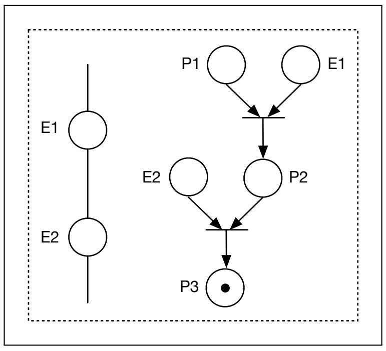


Figure 6.9: An example of a valid execution on a Petri net

Algorithm 5 The algorithm for reporting the failures

```
Require: $executedTransition$ (The transition that was executed from Algorithm 4)
Require: $RG$ (The reachability graph)
Require: Initialisations:
 1: $nextState \leftarrow$ null            # Variable to store the next state to $s$ state.
 2: $outputPlace \leftarrow$ null          # Variable to store the output place of the $executedTransisiton$.
 3: $report \leftarrow$ null               # Variable to store the debugging information.
 4: Function reportingModule
 5: Pass In: s:RG state, run:List          # Method parameters.
 6: Pass Out: report:String                # Method return value.
 7: $outputPlace \leftarrow executedTransition.getOutputPlace()$
 8: if $outputPlace$ is not a termination place then
 9:     if $s.getNextState()$ = null then
10:         $reporting \leftarrow + reportShortRun(execution.getInputPlace())$
11:     end if
12:     if $s.getNextState()$ != null then
13:         $reporting \leftarrow + reportMismatch(execution.getInputPlace(), run.getLastToken())$
14:     end if
15: end if
16: if $outputPlace$ is a termination place then
17:     $nextState \leftarrow s.getNextState()$
18:     if $nextState$ != null then
19:         $reporting \leftarrow + reportLengthyRun(execution.getInputPlace())$
20:     end if
21: end if
```

the plan graph expands (i.e., the depth of the plan graph increases). The plan graphs in Figure 6.8 depict Cases 12 and 13 from Table 6.1. The plan graph of Case 12 (see Figure 6.8) has four parallel branches (i.e., $b = 4$) and a depth of two (i.e., $d = 2$); it generates 2520 behaviour runs. By increasing the depth of the same plan graph to three (Case 13), the plan graph produces 369,600 behaviour runs. It is clear that the number of possible behaviour runs in a plan graph increases exponentially as the plan graph grows in terms of depth and branching.

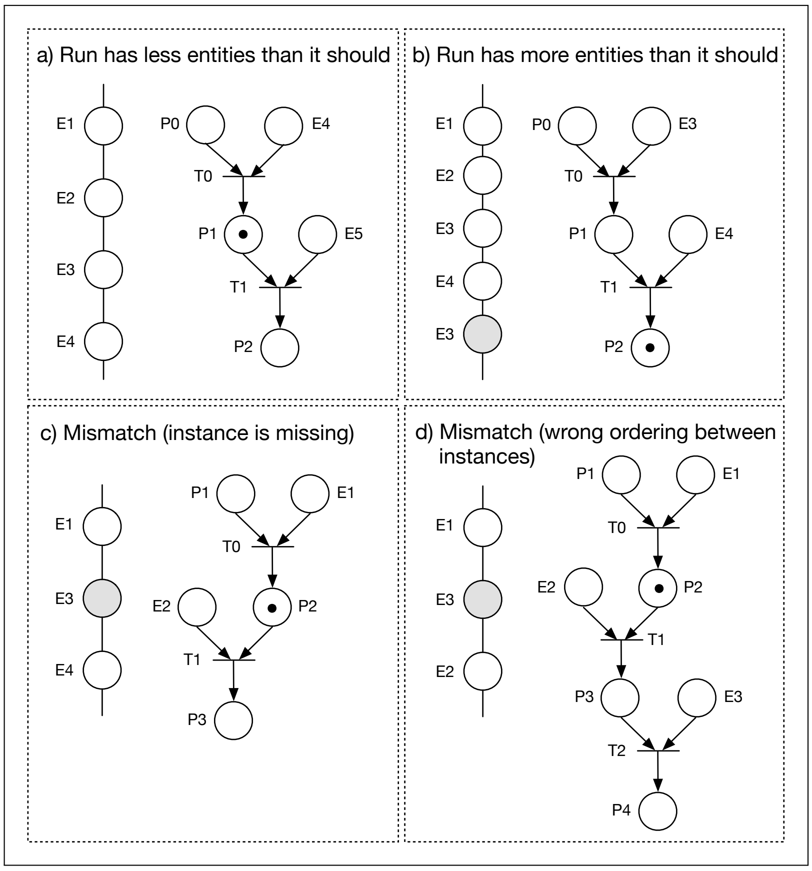

a) Run has less entities than it should
E1
E2
E3
E4
P0
E4
T0
P1
E5
T1
P2
b) Run has more entities than it should
E1
E2
E3
E4
E3
P0
E3
T0
P1
E4
T1
P2
c) Mismatch (instance is missing)
E1
E3
E4
P1
E1
T0
E2
P2
T1
P3
d) Mismatch (wrong ordering between instances)
E1
E3
E2
P1
E1
T0
E2
P2
T1
P3
E3
T2
P4

Figure 6.10: Examples on Petri net's execution failures

6.2 Fault Reporting

A behaviour run is reported as 'passed' if—and only if—the entire run is consumed and the execution of the Petri net reaches a termination place (see Figure 6.9). Otherwise, it is recorded as failed. To provide informative feedback about the agent-behaviour models, a cause for a failure is identified (see Table 3.1, page 73). In this section, we provide concrete examples of the different causes of failures.

As per **line 28** of Algorithm 3, the reporting module is called when the outcomes of the checking module in Algorithm 4 are *false*. The reporting module is detailed in Algorithm 5. As per **lines 9 to 21**, our approach categorises the failures into three categories, as follows:

The first failure occurs when the entire run is consumed, while the execution of the point-of-reference artefact Petri net is not at a termination place (see Figure 6.10 (a)). This failure indicates that the design **misses** the entity that is supposed to be realised according to the specified point-of-reference artefact. For example, the Petri net in Figure 6.10 (a) is expecting the ‘S5’ token so that the ‘T1’ transition can be fired, but the run is empty because the ‘S4’ token has already been consumed by firing the ‘T0’ transition. This failure, based on the previous example, highlights that the design part that is relevant to the point-of-reference artefact under investigation **misses** the ‘S5’ token.

The second type of failure happens when the run has more tokens than it should (i.e., failure 2 in Table 3.1, page 73). Such failures indicate that the design part that is relevant to the point-of-reference artefact under investigation has **repetitions** that the point-of-reference artefact does not require. Figure 6.10 (b) shows that the termination place ‘P2’ has a token, while the run has an ‘S3’ token to be consumed. This means that the relevant part of the design for the specified point-of-reference realises the ‘S3’ token twice, while the point-of-reference artefact, based on its Petri net, captures the ‘S3’ token once.

The final failure occurs when the execution of the Petri net is obstructed because of a mismatch between its current place in the Petri net and the current token from the run. This mismatch indicates that either the token is **missing** or that the **ordering** of tokens is wrong with respect to the point-of-reference artefact. For example, the failure to execute the Petri net in Figure 6.10 (c) results from a mismatch between the current token from the run and the current place in the Petri net. This is because the Petri net is expecting the ‘S2’ token, while the current token from the run is ‘S3’. Since ‘S2’ is not present in the run, then it is clear that the relevant parts of the design for the point-of-reference artefact miss the ‘S2’ token. However, in Figure 6.10 (d), although ‘S2’ is present in the run, the execution of the Petri net is still obstructed because the current token is ‘S3’ followed by the ‘S2’ token. This **wrong ordering** usually occurs when there is one plan in the agent-behaviour models that posts multiple entities, which are supposed to be posted sequentially.

In our reporting approach, we cluster the erroneous runs into the three categories mentioned above. Hence, executing 100,000 erroneous runs will not be reported as 100,000 errors. Instead, our approach groups the 100,000 errors into three categories (see Section 6.4). Further, to enhance the usability of the report, the module does not list all the erroneous runs. Instead, it lists one run, and reports how many runs capture similar errors (see Figure 6.13 in Section 6.4). This reporting is performed by tracking the design entity from the plan graph, which obstructs the execution and the final output place of the point-of-reference artefact Petri net. Then, the reporting module checks each faulty run and determines whether it is an instance of a previously-located fault.

6.3 Coverage Check

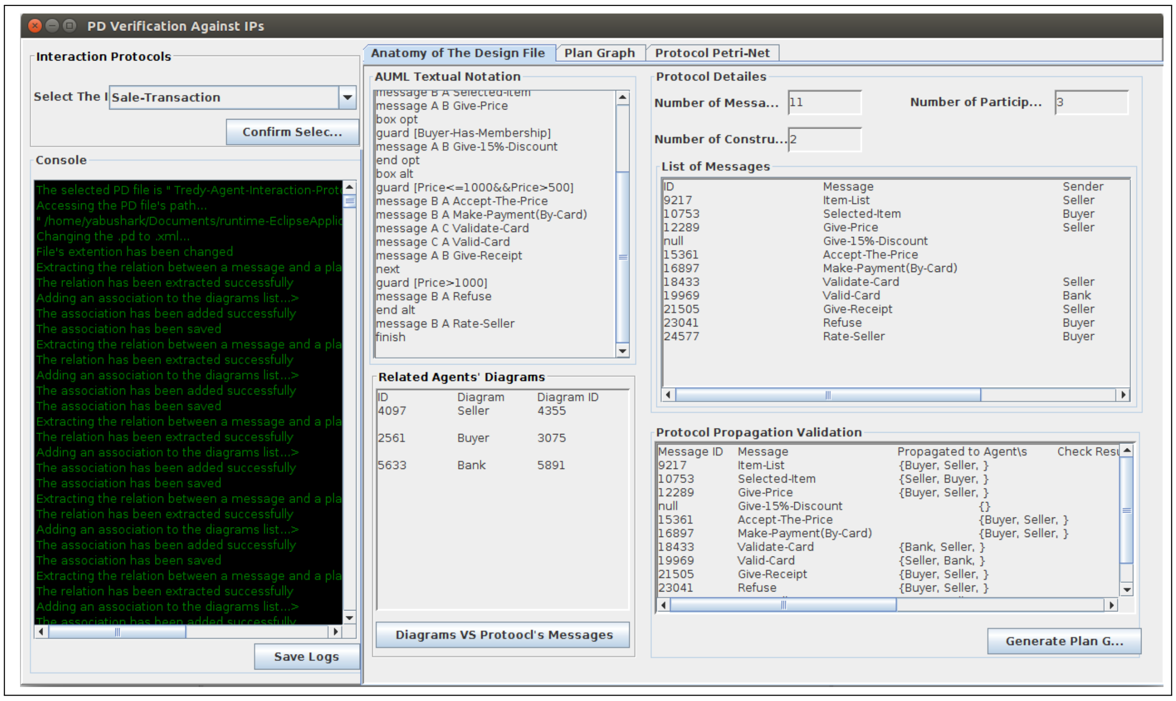
As explained in Section 3.4 (page 75), our verification framework allows software engineers to judge how the specified point-of-reference artefact is covered by the relevant parts of the agent-behaviour models. Our approach backtracks over the *unvisited* places of the Petri net for the specified point-of-reference artefact. Then, it constructs and reports full paths that include these unvisited places, indicating the percentage of coverage. This analysis assumes that the checking process results in a 100% pass rate. This is to ensure that the unvisited places of the Petri net for the specified point-of-reference artefact result from a lack of coverage and not from a defect in the design.

The coverage check is achieved by extracting all possible paths from the point-of-reference artefact Petri net using a traditional depth-first-search. It is important to note that this coverage check is meant to check whether the design covers all paths, not traces, of the specified point-of-reference artefact Petri net. A path is a set of places from the starting point of the Petri net to one of its termination places. However, a trace is a set of places that results from executing the Petri net. For example, a parallel fragment of two regions results in one path, whereas it results in two traces.

Although the consideration of all traces is a possibility, it is prohibitively costly and is likely to lead to many false-positives. *Node/arc* coverage is perhaps too easy, so the middle ground of ‘path’ coverage offers an apt trade off. Thus, in the case where the specified point-of-reference artefact has parallel control fragments, the check does not ensure that all possible traces are covered by the design under consideration. Instead, it verifies all paths, starting from the initial place and continuing until one of the termination places has been covered.

After extracting the paths out of the Petri net, our approach warns software engineers about the paths that contain unvisited places. Additionally, it reports the coverage percentage by calculating the ratio between the paths that contain unvisited places and the paths overall.

Assuming the reachability graph in Figure 6.2 results in a 100% passing rate, the protocol in Figure 6.4 has two possible sequences: one with the ‘Request’ message and the other without.



```
The selected PD file is " Tredy-Agent-Interaction-Prot
Accessing the PD file's path...
" /home/yabushark/Documents/runtime-EclipseApplic
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a pla
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a pla
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a pla
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a pla
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a pla
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
```

```
message B A Selected-Item
message A B Give-Price
box opt
guard [Buyer-Has-Membership]
message A B Give-15%-Discount
end opt
box alt
guard [Price<=1000&&Price>500]
message B A Accept-The-Price
message B A Make-Payment(By-Card)
message A C Validate-Card
message C A Valid-Card
message A B Give-Receipt
next
guard [Price>1000]
message B A Refuse
end alt
message B A Rate-Seller
finish
```

```
ID          Diagram     Diagram ID
4097        Seller      4355

2561        Buyer       3075

5633        Bank        5891
```

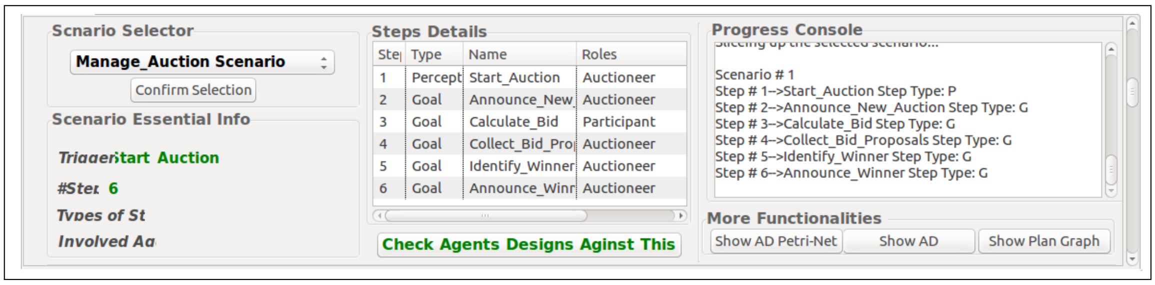
Number of Messa... 11 Number of Particip... 3

Number of Constru... 2

```
ID          Message                   Sender
9217        Item-List                 Seller
10753       Selected-Item             Buyer
12289       Give-Price                Seller
null        Give-15%-Discount
15361       Accept-The-Price
16897       Make-Payment(By-Card)
18433       Validate-Card             Seller
19969       Valid-Card                Bank
21505       Give-Receipt              Seller
23041       Refuse                    Buyer
24577       Rate-Seller               Buyer
```

```
Message ID  Message                 Propagated to Agent\s    Check Resu
9217        Item-List               {Buyer, Seller, }
10753       Selected-Item           {Seller, Buyer, }
12289       Give-Price              {Buyer, Seller, }
null        Give-15%-Discount                  {}
15361       Accept-The-Price                   {Buyer, Seller, }
16897       Make-Payment(By-Card)              {Buyer, Seller, }
18433       Validate-Card           {Bank, Seller, }
19969       Valid-Card              {Seller, Bank, }
21505       Give-Receipt            {Buyer, Seller, }
23041       Refuse                  {Buyer, Seller, }
```

Figure 6.11: The graphical user interface of the tool that verifies designs against interaction protocols



*Triager*Start Auction
#Ster 6
Types of St
Involved Aa

Ste	Type	Name	Roles
1	Percept	Start_Auction	Auctioneer
2	Goal	Announce_New	Auctioneer
3	Goal	Calculate_Bid	Participant
4	Goal	Collect_Bid_Pro	Auctioneer
5	Goal	Identify_Winner	Auctioneer
6	Goal	Announce_Winr	Auctioneer

```
Slicing up the selected scenario...

Scenario # 1
Step # 1-->Start_Auction Step Type: P
Step # 2-->Announce_New_Auction Step Type: G
Step # 3-->Calculate_Bid Step Type: G
Step # 4-->Collect_Bid_Proposals Step Type: G
Step # 5-->Identify_Winner Step Type: G
Step # 6-->Announce_Winner Step Type: G
```

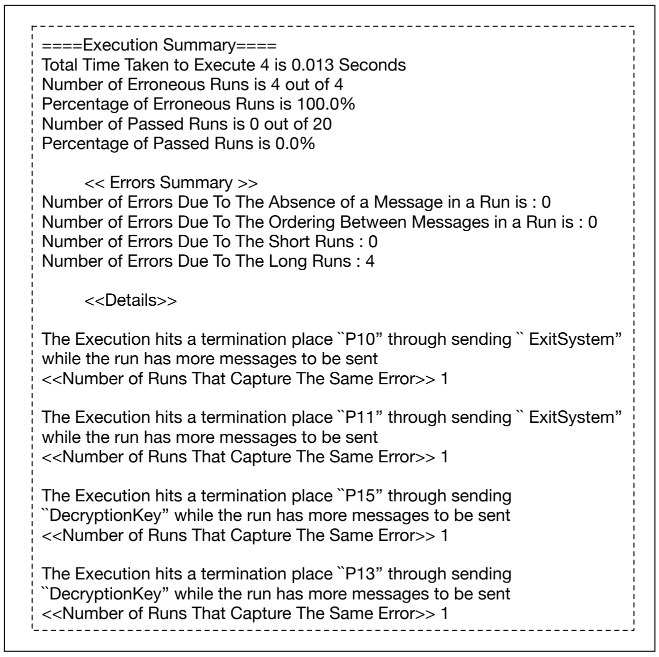
Figure 6.12: The graphical user interface of the tool that verifies designs against scenarios

Since the behaviours from the reachability graph cover both sequences, the coverage parentage of the design parts relevant to the 'Net-Bill' protocol is 100%.

6.4 Tool Support

We have implemented an Eclipse plug-in[1] that extends the Prometheus Design Tool (PDT) to automate our approach (see Figures 6.11 and 6.12 for screenshots). The tool takes the PDT design file as an XML file and generates the necessarily verifiable components. First, it transforms the

[1] https://tinyurl.com/PDT-verificationTool



```
====Execution Summary====
Total Time Taken to Execute 4 is 0.013 Seconds
Number of Erroneous Runs is 4 out of 4
Percentage of Erroneous Runs is 100.0%
Number of Passed Runs is 0 out of 20
Percentage of Passed Runs is 0.0%

        << Errors Summary >>
Number of Errors Due To The Absence of a Message in a Run is : 0
Number of Errors Due To The Ordering Between Messages in a Run is : 0
Number of Errors Due To The Short Runs : 0
Number of Errors Due To The Long Runs : 4

        <<Details>>

The Execution hits a termination place ``P10" through sending `` ExitSystem"
while the run has more messages to be sent
<<Number of Runs That Capture The Same Error>> 1

The Execution hits a termination place ``P11" through sending `` ExitSystem"
while the run has more messages to be sent
<<Number of Runs That Capture The Same Error>> 1

The Execution hits a termination place ``P15" through sending
``DecryptionKey" while the run has more messages to be sent
<<Number of Runs That Capture The Same Error>> 1

The Execution hits a termination place ``P13" through sending
``DecryptionKey" while the run has more messages to be sent
<<Number of Runs That Capture The Same Error>> 1
```

Figure 6.13: Text-based verification report

specified point-of-reference artefact to a Petri net. Second, the tool generates a plan graph that merges the relevant parts to the specified point-of-reference artefact from the agent-behaviour models. Then, the conformity of the behaviour runs from the plan graph are checked against the Petri net of the specified point-of-reference artefact. The output of our tool is a report that provides:

1. Detailed logs of the erroneous runs, including their categorisation (see Table 3.1 on page 73)
2. An execution summary. The summary includes information such as the execution time, the number of behaviour runs that passed and those that failed.

Figure 6.13 depicts a snapshot of the generated report for the checking process of the 'Net-Bill' protocol. The 14 behaviour runs were extracted from the reachability graph. After executing all the runs against the protocol's Petri net, 14 out of 14 runs were erroneous. These errors fell into two types of errors: (1) runs that contain more messages than they should; (2) incorrect ordering between messages within runs.

Summary

This chapter details the technicalities of checking the conformity of runs derived from the agent-behaviour models with the specified point-of-reference artefact Petri net. First, our approach takes the Petri net equivalent to the plan graph of the agent-behaviour models as an input. Then, it calculates the reachability graph of the Petri net. Since the reachability graph captures all possible ways of executing the Petri net, behavioural runs that cover all possible combinations can be extracted by traversing the reachability graph. Thus, we use a stranded *depth-first-search* algorithm to extract all possible runs from the reachability graph.

The correctness check is performed by running the possible behavioural runs extracted from the reachability graph over the point-of-reference artefact Petri net and logging any violations. To maintain the computational complexity of the approach, especially in situations where the reachability graph includes too many behavioural runs, the approach considers one token at a time.

The output of our verification framework is a text-based report that documents information about the design under investigation. We implement this verification approach as an Eclipse plug-in that integrates with the Prometheus Design Tool (PDT).

CHAPTER 7

Evaluation

This thesis has proposed a method that uses design artefacts to verify agent-based designs. In its application, both agent-behaviour models and the specified point-of-reference artefacts were framed structurally as executable structures and compared against each other to identify inconsistencies. As such, this chapter presents the empirical evaluation of the verification framework proposed in this thesis by:

1- Evaluating the verification framework with respect to its effectiveness for detecting defects (see Section 7.2).

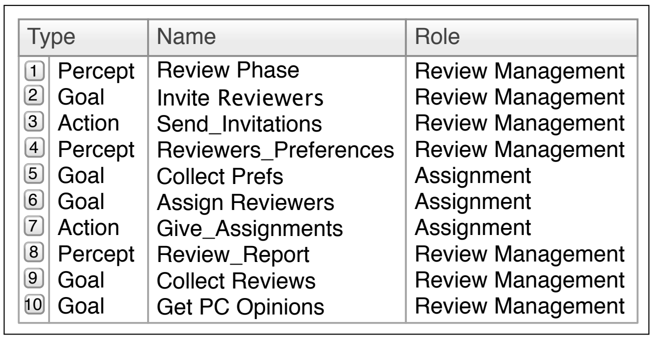
2- Analysing the scalability of the approach as the number of goals and plans in the design grows (see Section 7.3).

7.1 Evaluation Objective

The goals of this evaluation are:

1. To assess whether the proposed mechanism can detect defects in agent designs with respect to the specified point-of-reference artefact.

2. To determine the level of false positives generated by the proposed approach.



Type	Name	Role
1 Percept	Review Phase	Review Management
2 Goal	Invite Reviewers	Review Management
3 Action	Send_Invitations	Review Management
4 Percept	Reviewers_Preferences	Review Management
5 Goal	Collect Prefs	Assignment
6 Goal	Assign Reviewers	Assignment
7 Action	Give_Assignments	Assignment
8 Percept	Review_Report	Review Management
9 Goal	Collect Reviews	Review Management
10 Goal	Get PC Opinions	Review Management

Figure 7.1: Review scenario description

3. To determine whether the time taken to run experimental tools is reasonable[1] considering the complexity of the specified point-of-reference artefact and its agent-behaviour models.

7.2 Evaluating The Effectiveness

In this section, we empirically show the effectiveness of the verification approach proposed by this thesis. We use the tool-support explained in Section 6.4 on page 168 to facilitate the evaluation. Our tool-support accesses and tokenises the contents (i.e., the scenarios, protocols, goal models and agent-behaviour models) of the specified design file. In evaluating the effectiveness of our approach, we consider two sets of evaluations: (1) a mutation evaluation and (2) an empirical evaluation of the implemented systems.

7.2.1 Effectiveness via Mutation

This section presents a preliminary assessment to validate our approach and its ability to detect defects in a given design and to facilitate learning about the types of problems it cannot detect.

Method

We use the ‘conference-management system’ case study in this evaluation [Padgham et al. 2007]. This system helps to manage the different phases of the conference review process, including submission, review, decision and paper collection. In the submission phase, the system should be able to assign a number to each submission and provide receipts to authors. After the specified submis-

[1] ‘Reasonable’ is defined as performing a complete execution within 24 hours on an industry-scale system

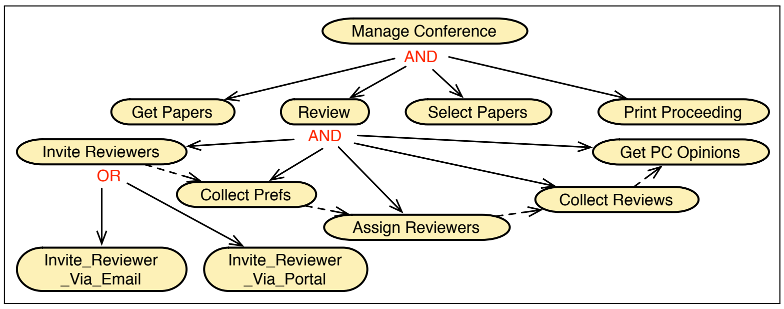


Figure 7.2: Goal diagram for the conference-management system

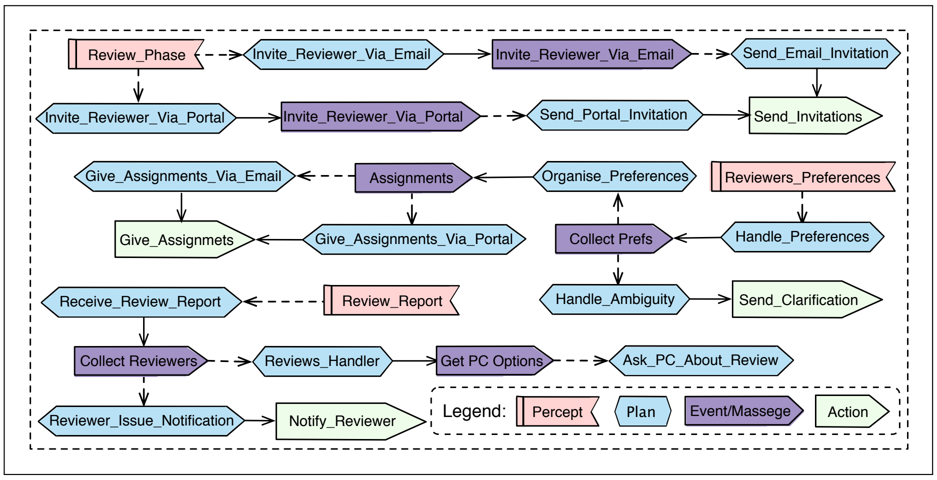


Figure 7.3: Detailed design for the *'Review Manager'* agent based on the *'Review'* scenario

sion deadline, the system assigns papers to reviewers, who review the paper. After receiving the reviews, the system supports making decisions on whether to accept or reject each paper, notifying the authors. Then, the system collects the accepted papers and prints them as conference proceedings. Figure 7.1 shows a scenario for the conference-management system, while Figure 7.2 shows the goal-overview diagram for the system.

We designed the 'Review Manager' agent involved in the 'Review' scenario (see Figure 7.3). Then, using the tool support, we generated the plan graph equivalent to the agent's behaviour model (see Figure 7.4). Figure 7.4 shows the plan graph corresponding to the design of the *Review Manager* agent, which fulfils the *Review* scenario.

Two external participants who were experienced in BDI-agent design were given the plan graph in Figure 7.4. Each participant was asked to make several small changes (called mutations), such as adding, removing, replacing and renaming entities. In total, we received 11 mutations. Note that the mutations on the plan graphs were manually inputted (using *paper and pen*), as the

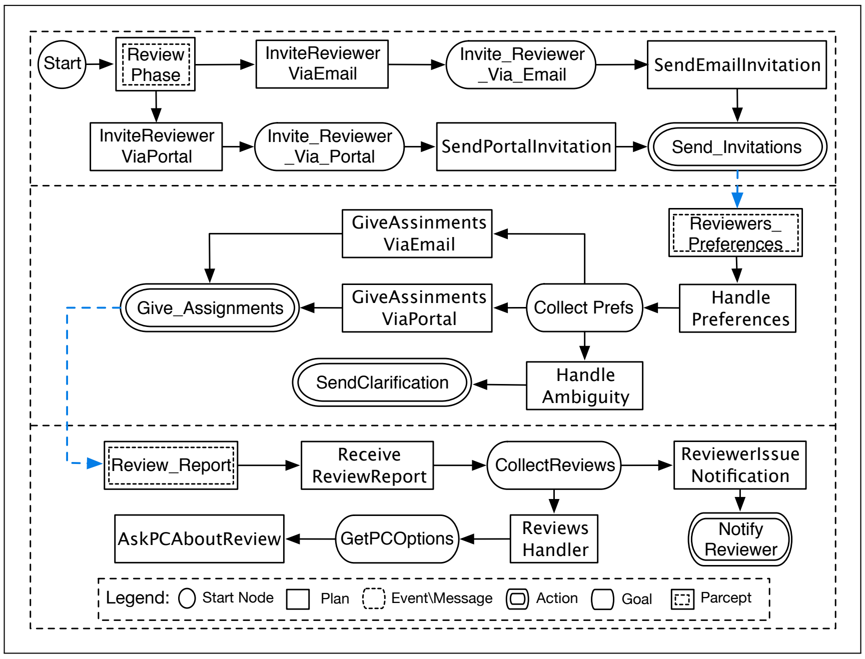


Figure 7.4: Plan graph for the *'Review Manager'* agent based on the *'Review'* scenario

Category	**ID**	**MS**	**TS**	**TL**	**OE**	**Def.**	**Det.**
ADD	MA0	0	0	1	0	✔	✔
	MA1	0	0	0	0	✔	✘
	MA2	0	0	0	0	✔	✘
	MA3	0	0	0	0	✔	✘
	MA4	0	0	0	0	✔	✘
DEL	MR0	0	1	0	0	✔	✔
	MR1	0	4	0	0	✔	✔
	MR2	0	2	0	0	✔	✔
	MR3	0	0	0	0	✘	✘
REP	MP0	0	0	0	0	✘	✘
	MP1	0	0	0	0	✘	✘
Total	11	0	7	1	0	8	4

Table 7.1: Summary of potential problems raised by the plan graph mutation per mutations category (ID: mutation ID,MS: missing steps, TS: run too short, TL: run too long, OE: miss-ordering between steps; Def.: modified design contains a defect with respect to the scenario from the mutator prospective; Det.: the defect is detected.)

tool support provides a static digital view of plan graphs. We then digitally reproduced the design as per each mutated plan graph version. The tool support assistance was applied to each design—each of which was examined as to whether the introduced (known) defects were detectable.

Results

Although the mutations may have fallen into more than three categories, for this study we coded the mutations into three variations. First, the addition of new (irrelevant) entities *(ADD)*. As an example, one participant added a new goal between the *Assign Reviewers* and *Collect Prefs* steps. Further, they made the plan that posts the 'Assign Reviewers' goal post a new goal: 'Optimise Allocation'. Subsequently, new plans were created to accommodate the 'Optimise Allocation' and the 'Assign Reviewers' goals.

The second coded mutation involved the deletion *(DEL)* of entities from plan graphs and the renaming of them inconsistently and out of context with respect to the conference scenario.

Third, there was a replacement of goal steps *(REP)* with related goals (parents or children) from the goal tree.

Table 7.1 summarises the results obtained by applying our approach on the 11 mutations. Only one mutation (MA0) out of the five in the *ADD* category was detected. This was a result of the introduction of a new goal, creating a new step in the scenario that had not been originally specified. The remaining four mutations added new functionality to the design that was filtered because it was not part of the scenario (recall that behaviour runs have elements that do not occur in the scenario filtered out). Such a filtering process takes place in our verification approach because we check one point-of-reference artefact at a time. However, in future research, we will investigate how to consider multiple point-of-reference artefacts at once to address such cases.

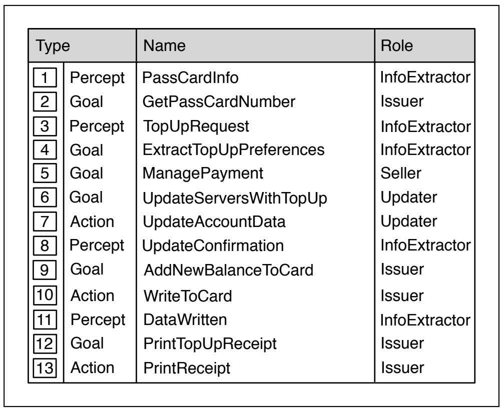
In the *DEL* category, three of the four mutations were detected. For the undetected mutations, the participant modified the plan graph such that the goal *Assign Reviewer* was handled by one plan instead of two. This simply reduced the number of (correct) runs and, hence, the design was still correct with respect to the 'Review' scenario.

The *REP* mutations did not produce defects in the design and, consequently, the tool did not find defects in the design. Participants replaced goals in the design with child/parent goals from the goal hierarchy that were valid with respect to the scenario.

This shows that our approach is promising, despite the small numbers involved. In particular, if we consider our technique as a method of classifying mutations as correct or incorrect, we correctly identified eight out of 11 mutations as either defective or correct. The four incorrectly

Cases Category	Number of Designs	Total Number of Scenarios	Total Number of Protocols
Students Assignments	7	7	7
Expert Designs	6	0	6
Case Studies	2	3	9
Total	15	10	22

Table 7.2: Objects of analysis



Type		Name	Role
$\boxed{1}$	Percept	PassCardInfo	InfoExtractor
$\boxed{2}$	Goal	GetPassCardNumber	Issuer
$\boxed{3}$	Percept	TopUpRequest	InfoExtractor
$\boxed{4}$	Goal	ExtractTopUpPreferences	InfoExtractor
$\boxed{5}$	Goal	ManagePayment	Seller
$\boxed{6}$	Goal	UpdateServersWithTopUp	Updater
$\boxed{7}$	Action	UpdateAccountData	Updater
$\boxed{8}$	Percept	UpdateConfirmation	InfoExtractor
$\boxed{9}$	Goal	AddNewBalanceToCard	Issuer
$\boxed{10}$	Action	WriteToCard	Issuer
$\boxed{11}$	Percept	DataWritten	InfoExtractor
$\boxed{12}$	Goal	PrintTopUpReceipt	Issuer
$\boxed{13}$	Action	PrintReceipt	Issuer

Figure 7.5: Top up scenario description

classified mutations were a result of the filtering process. It is also encouraging to note that there were no false positives (i.e., all detected defects corresponded to actual design defects).

7.2.2 Effectiveness via Application Systems

In this section, real cases were used to test for defects within designs. Table 7.2 shows the details of the object of analysis we used for this effectiveness evaluation. We verified the validity of 15 agent-behaviour models against 32 point-of-reference artefacts in total. We have categorised these 15 objects of analysis into three categories (see Table 7.2):

Category 1: student assignments — As part of an RMIT agent-oriented programming and design course, students were required to design a 'Top-up' scenario (see Figure 7.5) for the 'smart-kiosk ticketing' system detailed in Section 2.4 on page 22. The 'Top-Up' specifications are as follows:

'***Top-up Description***: the system should allow customers to recharge their pass cards based on the type of card (pass or money). After the pass card is touched onto the RFID reader/writer

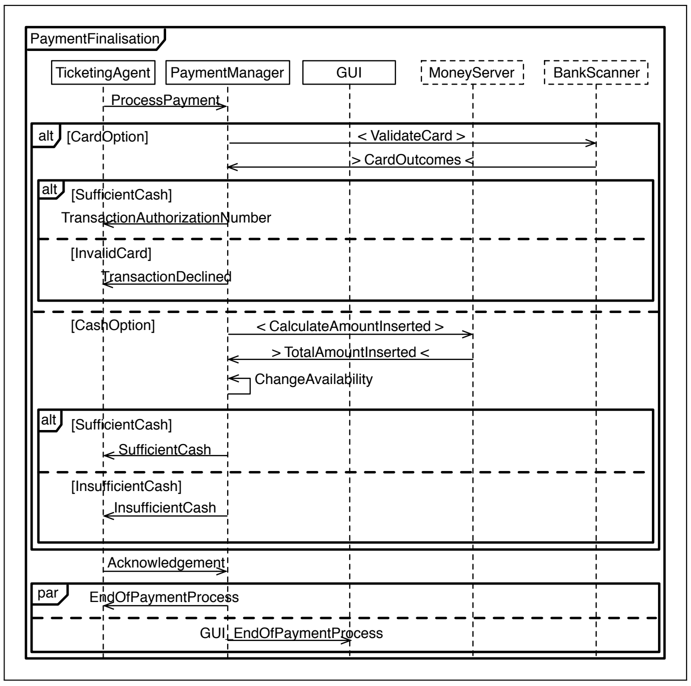


Figure 7.6: Payment finalisation AUML protocol

port, the system should be able to recognise the card and obtain the card number. Through the touch screen, customers should be able to tap on the 'Top-Up' option. Then, the system should allow customers to specify, through the touch screen, the type of card (money or pass), the top-up amount and the payment method (card or cash). After finalising the payment process (through the protocol explained below), the system should update both the provider servers and the pass card (using the RFID reader/writer).

The payment process involves an interaction between multiple participants (see Figure 7.6) including a Ticketing-Agent, Payment-Manager, GUI, Money-Scanner (actor) and Banking-Server (actor). This payment process is different based on the chosen payment option. If the card option is chosen, the process involves the following: verifying the card with the banking server, deducting the amount from the customer's account and then the provision of a transaction number by the banking server. Then, the system should push such updates over to the company's servers. If cash is the option chosen, then the system should recognise the inserted amount and decide the requisite change for the customer. As a final step of the top-up transaction, customers should be notified about the outcome of the transaction (whether it has been approved or not). If the transaction has been approved, the notification

	Design part related to scenario			Design part related to protocol		
Design	#Plans	#Entities	Total	#Plans	#Entities	Total
1	13	21	34	14	15	29
2	5	8	13	13	14	27
3	13	15	28	14	15	29
4	4	7	11	15	13	28
5	4	7	11	11	13	24
6	18	23	41	15	16	31
7	9	15	24	12	19	31

Table 7.3: Number of design units for the first category, students assignments (#Entities: number of events, actions and percepts

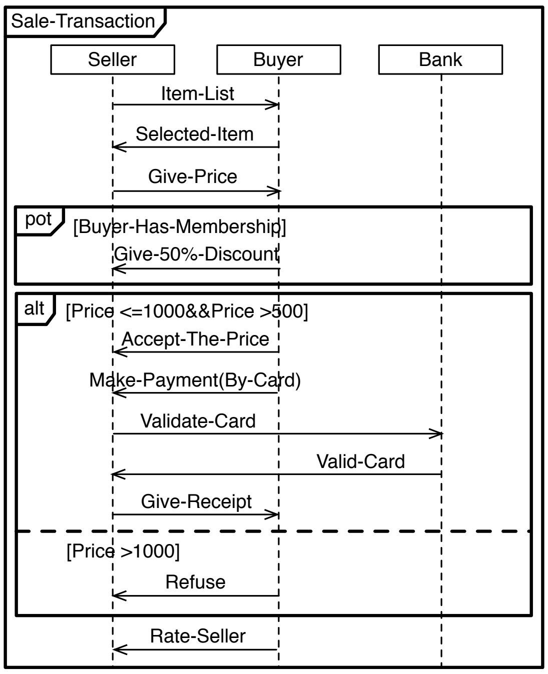


Figure 7.7: Sale transaction AUML-sequence diagram

should be provided in the form of a printed receipt.'

As per Table 7.2, this category includes seven different designs and each design implements one scenario and one protocol. We checked each design against both the scenario and the protocol. Details about the number of plans and other design entities in these agent models are summarised in Table 7.3.

Category 2: expert designs — Three interaction protocols that were used in this evaluation were

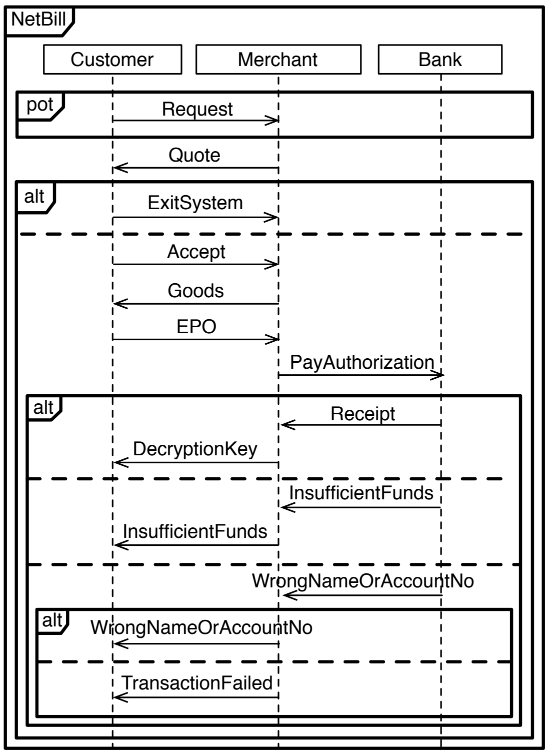


Figure 7.8: Secure Net-Bill AUML-sequence diagram

‘Sale-Transaction’, the ‘Net-Bill’ protocol and the ‘Secure-Net-Bill’. We investigated three designs for the ‘Sale-Transaction’ protocol, two for the ‘Net-Bill’ and one for the ‘Secure-Net-Bill’. Each design was produced by a person outside of our research team who was familiar with BDI modelling and the Prometheus methodology. All participants had experience, ranging from three to 15 years in the agent field, specifically, in the Prometheus methodology.

Figure 7.7 shows the AUML protocol for the ‘Sale-Transaction’ system, which was designed by us. The corresponding system models an online store as a multi-agent system with three agents (the ‘Seller Agent’, the ‘Buyer Agent’ and the ‘Bank Agent’) that interact with each other. We asked three participants to complete the behaviour models for each of the agents involved in the protocol, resulting in three different designs.

A complete design of the agent system following the ‘Net-Bill’ protocol (see Figure 5.15 on page 129) was produced by two experts in the field. In addition, a group of researchers, at the University of Melbourne with extensive experience in Prometheus methodology restricted

Design	Design part related to protocol #Plans	#Entities	Total
Sale-Transaction 1	17	15	32
Sale-Transaction 2	11	8	19
Sale-Transaction 3	13	12	25
Net-Bill 1	13	9	22
Net-Bill 2	12	9	21
Secure Net-Bill	12	17	29

Table 7.4: Number of design units for the second category, expert designs. (#Entities: number of events, actions and percepts

Protocol Name	#Protocol's Entities	Design part related to protocol #Plans	#Entities	Total
ATS	37	21	30	51
Prepare-Arrival	5	6	5	11
Handle-Baggage	4	4	3	7
Board	4	4	3	7
De-board	4	3	2	5
Prepare-Departure	6	7	6	13
Maintain-Aircraft	3	4	3	7
Service-Aircraft	4	4	4	8

Table 7.5: Number of design units for the Aircraft Turnaround Simulator. (#Entities: number of Events, Actions and Percepts

this protocol to enhance its security. Figure 7.8 depicts the secure version of the 'Net-Bill' protocol. Then, they produced the agent-behaviour models according to the role of each agent in the protocol.

In total, we investigated six designs against six interaction protocols. Table 7.4 details these six agent models with respect to the number of plans and other design entities.

Category 3: case studies — Two case studies were used in the evaluation: (1) aircraft turnaround simulation (ATS)[2] and (2) oil production simulation[3]. The ATS case study consisted of joint work between the University of Melbourne and its industry partner, Jeppesen [Miller et al. 2014]. Briefly, the system simulates the operations that take place at the airport when an aircraft lands. The system implements eight protocols, which model the interactions

[2]Refer to Appendix C for the protocols used in this case study and the agent-behaviour models.
[3]Refer to Appendix D for the protocols used in this case study and the agent-behaviour models.

	Scenario Information				Design part related to Scenario		
Scenario	#Goals	#Actions	#Percepts	Total	#Plans	#Entities	Total
ControlSendContent	5	1	1	7	14	16	30
RespondToOperator	1	1	1	3	5	5	10
OptimiseOilProduction	5	1	1	7	23	24	47

Table 7.6: Details of the scenarios in the oil production simulation.

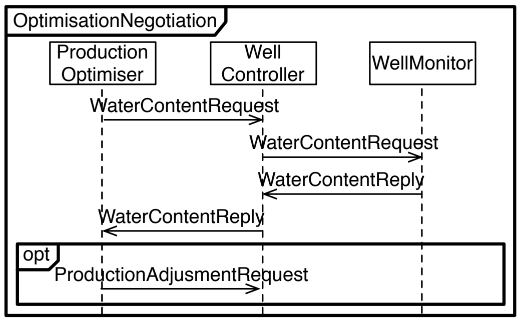


Figure 7.9: Optimisation negotiation AUML-sequence diagram

between 11 agents. The largest of these eight protocols involves all 11 agents with 37 design entities. This protocol is of particular interest for us in assessing the types of errors we would find because it was constructed early in the project and not maintained as the project evolved and, thus, some changes to other models may not have been reflected in the protocol. Table 7.5 lists details of the relevant parts for the design to each protocol.

The other case study (oil production simulation) was developed in the context of a master's research project [Engmo and Hallen 2007]. The system is an agent-based simulation that simulates the operations and processes of a simple oil field. The system includes six agents, who participate in three scenarios and one protocol. As Figure 7.9 depicts, the agents in the 'Optimisation Negotiation' protocol exchange five messages and the relevant design parts of this protocol capture six plans that are associated with the five messages. The three scenarios are concerned with managing the oil field. Table 7.6 provides an overview of these three scenarios.

The Experimental Procedure

Fifteen designs were used in this evaluation. These designs are divided into three categories and each category is configured as follows:

Category 1: student assignments — As per the assignment specifications, students were asked to work in pairs and to use PDT (Eclipse Plug-in Version 0.4) when designing the system. Additionally, they were notified that their design must conform to the order that enforced by the specifications of the 'Top-up' transaction and the interaction protocol ('Payment-Finalisation' shown in Figure 7.6). They were told to ensure comprehensive descriptors for each design entity. More importantly, students were advised that if a plan posted multiple events, the relationship between these events was to be assumed to be parallel.

We investigated ten assignments and excluded the ones that lacked comprehensive descriptors for design entities. We filtered three assignments out of the evaluation because they lacked information (i.e., no descriptors for the design entities). Thus, the evaluation included seven assignments out of the ten initial submissions.

Category 2: expert designs — Each participant was given specifications for their assigned protocol. They were then asked to design the agents according to their roles in the protocol. Participants were given a laptop with PDT (Eclipse Plug-in Version 0.4) installed and were asked to design the protocol using PDT, documenting their design assumptions via the descriptor of every plan they created. Unlike in the previous category, participants were not restricted by the parallelism assumption. In other words, they were free to design the choice through having one plan to post multiple events, assuming the *selection* statement would be coded into the plan's body to post only one event. We asked participants to document their assumptions.

Category 3: case studies — For the two cases under this category, we undertook the following:

We reverse-engineered the Jason implementation of the ATS system into a PDT design. We followed a one-to-one mapping approach. In Jason, each function represents a plan that has an input and an output. From the code, we can distinguish between the different design entities (i.e., messages, events, goals, actions and percepts).

For the oil production simulator, we manually created the PDT design file from the master's thesis project. We obtained the design artefacts from the appendices of the thesis [Engmo and Hallen 2007].

For all the three categories, we manually checked the consistency of the design entities' names against the specified point-of-reference artefact. We then followed an iterative process for checking each design. This iterative process involved the following three steps for each design:

1. *Execution:* We ran our tool over the PDT design file (including both the point-of-reference artefact and the agent-behaviour models) to produce a report. We used a laptop running a 64-bit Intel Core i7 processor clocked at 3.1 GHz. 1 GB of RAM was dedicated to utilisation by the Java Virtual Machine.

2. *Inspection:* Using the report generated, we analysed the causes of failed behaviour runs, categorising each cause as either a true positive, a false positive or an unknown. A false positive is a warning raised that we believe not a defect in the design but is caused by a clear and valid assumption made by the designer. An unknown categorisation implies that the design produces runs that fail, perhaps because the designer made explicit design assumptions that we do not understand.

3. *Modification:* Some defects mask the presence of other defects. As such, we modified the design in such a way as to rectify the causes of the reported failures, including false positives and unknowns. The changes involved adding, removing and modifying plans, other design entities (i.e., events, actions, percepts and goals) and associations.

We iterated each design until our tool reported no problems. During each iteration, we recorded the following:

1. The number of warnings raised by the tool.

2. The number of true positive, false positive and unknown defects.

3. The time that was taken to extract the runs from the reachability graph.

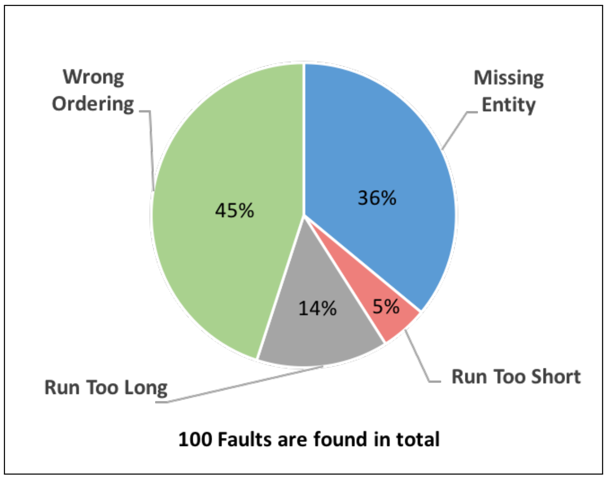


Figure 7.10: Distribution of faults over all iterations

4. The level of coverage the design offered concerning the specified point-of-reference artefact whether the design accomplished 100% passing rate.

5. The number of extracted behaviour runs.

Results

This section presents the results of the evaluation outlined in Section 7.2.2. As per the iterative process described, we were able to identify and categorise the total number of defects found (true positives), the number of false positive defects found and the number of potential defects that could not be accurately categorised for each of the designs. Further, we identified the level of coverage offered by each design with respect to the point-of-reference artefact under investigation.

Overall, we examined 15 designs against 32 point-of-reference artefacts via 104 iterations and discovered 100 potential problems. Figure 7.10 provides an illustration of the different categories of errors as applied to the 100 faults that were raised (see Table 3.1 on page 73). As the figure demonstrates, almost a half of the failures resulted from the wrong ordering (MO) of design entities with respect to the specified point-of-reference artefact, while just a third of errors arose from missing entities (ME). A small fraction of behaviour runs was unsuccessful as a consequence of fewer entities in the run (TS) according to the specified point-of-reference artefact, while 14 faults out of 100 resulted from runs that had more entities than they should (i.e., TL).

Based on the designers' assumptions, we divided the 100 faults into three categories: (1) true

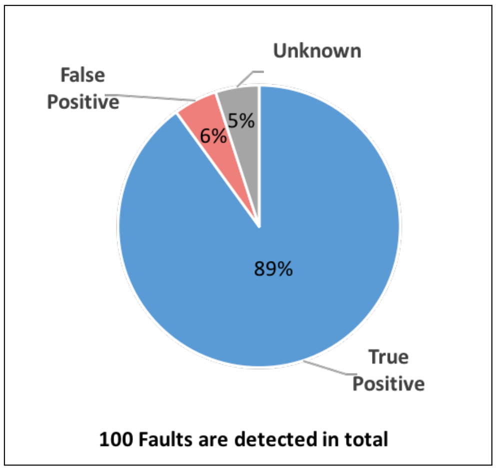


Figure 7.11: Categorisation of defects over all iterations

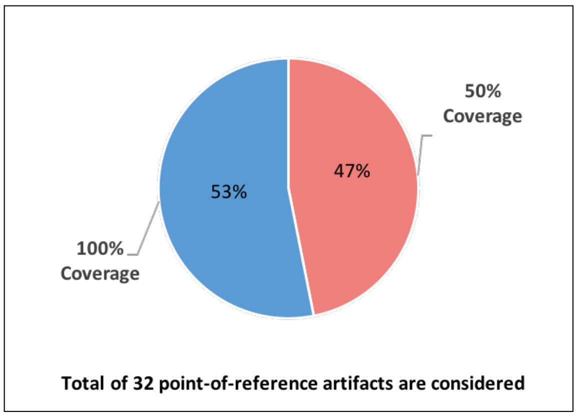


Figure 7.12: Coverage check over all cases

positive, (2) false positive and (3) unknown. Figure 7.11 visualises the categorisations of faults for the 104 iterations. True positives imply actual design defects, while false positives and unknown faults were flagged as warnings to the designers. As can be seen from the figure, we were able to detect 89 defects in 32 design parts with only six false positives and five faults that were unable to be categorised. This provided a strong indication that our approach, via the tool-support, indeed detects defects in agent-behaviour models with respect to the specified point-of-reference artefact, producing few false positives and in a reasonable amount of time, as we demonstrate later in this chapter.

The tool support reported the level of coverage of each valid design with respect to the specified point-of-reference artefact at the end of each experimental run. Similar to the false positives and the unknown faults, as mentioned above, the coverage percentage was flagged as a warning

	Design Against Scenario				Design Against Protocol			
ID	True Positive	False Positive	Unknown	Total	True Positive	False Positive	Unknown	Total
D 1	14	0	0	14	3	0	0	3
D 2	5	0	0	5	3	0	0	3
D 3	3	0	0	3	4	0	0	4
D 4	6	0	0	6	1	0	0	1
D 5	6	0	0	6	2	0	0	2
D 6	7	0	0	7	5	0	0	5
D 7	3	0	0	3	3	0	0	3
Total	44	0	0	30	21	0	0	21

Table 7.7: Categorisation of defects over all iterations for student assignments category

	Design Against Scenario							Design Against Protocol						
ID	#Iter	ME	TS	TL	MO	Total	Cov	#Iter	ME	TS	TL	MO	Total	Cov
D 1	15	0	0	1	13	14	100	4	0	0	0	3	3	100
D 2	6	5	0	0	0	5	100	4	0	0	2	1	3	100
D 3	4	2	0	0	1	3	100	5	0	0	1	3	4	100
D 4	7	6	0	0	0	6	100	2	0	1	0	0	1	100
D 5	7	6	0	0	0	6	100	3	1	0	1	0	2	100
D 6	8	0	0	2	5	7	100	6	1	2	1	1	5	50
D 7	4	2	0	0	1	3	100	4	1	1	0	1	3	50
Total	51	21	0	3	20	44	—	28	3	4	5	9	21	—

Table 7.8: Summary of potential problems raised by the tool concerning the students assignments category (#Iter: number of iterations, ME: missing entity, TS: run too short, TL: run too long, MO: mismatch between entities, Cov: percentage of coverage)

sign. The findings of the coverage check that we conducted on 32 point-of-reference artefacts are shown in Figure 7.12. Over one third of the cases resulted in a partial coverage, while 53% of the cases resulted in 100% coverage. Although a large proportion of the cases resulted in 100% coverage, most of these cases were related to scenarios with only one basic run and without variations.

In the above, an overview analysis of the results we obtained from evaluating our verification framework via the implemented systems. In what follows, we break down our results based on the three categories of the systems used in this evaluation.

Category 1: student assignments — Since students were required to document all the design

	Design Against Scenario		Design Against Protocol	
ID	#Behaviour Runs	Time Taken (in seconds)	#Behaviour Runs	Time Taken (in seconds)
D 1	2	0.001	8	0.001
D 2	1	< 0.001	8	0.003
D 3	2	< 0.001	8	< 0.001
D 4	1	< 0.001	8	0.001
D 5	1	< 0.001	8	0.001
D 6	1	< 0.001	4	0.001
D 7	6	0.003	4	0.001

Table 7.9: Time Analysis of Runs Concerning the Student Assignments Category.

assumptions they made, all faults in executing the behaviour runs against the Petri net into both the ‘Top-Up’ scenario and the ‘Payment-Finalisation’ protocol represent defects in the agent-behaviour models. Table 7.7 shows that all failures in behaviour runs against the Petri net for both the ‘Top-Up’ scenario and the ‘Payment-Finalisation’ protocol are true positives.

Table 7.8 demonstrates the information about the defects that were found in the seven assignments considered in the evaluation, concerning both the ‘Top-Up’ scenario and the ‘Payment-Finalisation’ protocol. At the end of each iteration, a defect was addressed. For example, in ‘D 1’, the wrong ordering between the first percept in the ‘Top-Up’ scenario (‘Top-Up-Request’) and another goal step was fixed after the first iteration. The sequence of the scenario states that the ‘Top-Up-Request’ Percept needed to be sent after posting the ‘GetPassCardNumber’ goal. However, the design had the plan that posted the ‘GetPass-CardNumber’ goal posted another goal related to the scenario (i.e., it was included in the run). Thus, the behaviour run underwent the goal step before the percept. We changed the sequence of the design to ensure that the percept was posted immediately after the intended goal to address this ordering issue. After 15 iterations, the ‘D 1’ design was free of defects concerning the scenario; at this point, we iterated the design four times to ensure its validity in relation to the protocol.

Overall, we iterated all the designs 79 times over to address 65 defects detected with respect to the ‘Top-Up’ scenario and the ‘ Payment-Finalisation’ protocol. The majority of

Case ID	True Positives	False Positives	Unknown	Total
Sale-Transaction 1	3	0	2	5
Sale-Transaction 2	1	0	0	1
Sale-Transaction 3	1	2	0	3
Net-Bill 1	0	1	0	1
Net-Bill 2	0	0	0	0
Secure Net-Bill	2	2	0	4
Total	7	5	2	14

Table 7.10: Categorisation of defects over all iterations for the expert designs category

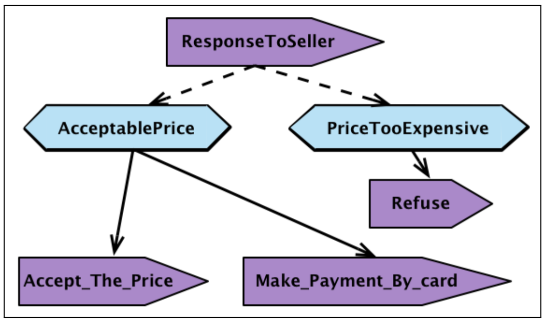


Figure 7.13: Buyer Agent overview diagram

defects fell into two categories: (1) the *wrong-ordering* category (more than 40%) and (2) the *missing entity* category (39%).

Regarding the point-of-reference coverage, all seven designs fully cover the ‘Top-Up’ scenario, as the scenario does not have variations. However, two designs resulted in a 50% coverage of the ‘Payment-Finalisation’ protocol. Neither design ‘D 6’ nor ‘D 7’ had a run that covered the ‘cash option’ region captured by the protocol.

Table 7.9 demonstrates the time-cost for checking the conformity of the runs from the reachability graph against both the scenario and the protocol Petri nets. Note that the table lists the *worst-case scenario* (i.e., the entire reachability graph was traversed). The time-cost is low for all the cases, at almost zero seconds.

Category 2: expert designs — Unlike the first category above, not all the design assumptions were documented. Hence, not all the faults found could be classified as true positives. As Table 7.10 shows, we were able to detect seven defects in six different designs with only

Case ID	#Iter	ME	TS	TL	MO	Total	Cov
Sale-Transaction 1	6	1	0	1	3	5	100
Sale-Transaction 2	2	1	0	0	0	1	50
Sale-Transaction 3	4	0	1	0	2	3	50
Net-Bill 1	2	0	0	1	0	1	50
Net-Bill 2	1	0	0	0	0	0	50
Secure Net-Bill	5	0	0	3	1	4	100
Total	20	2	1	5	6	14	—

Table 7.11: Summary of potential problems raised by the tool concerning the expert designs category (#Iter: number of iterations, ME: missing entity, TS: run too short, TL: run too long, MO: mismatch between entities, Cov: percentage of coverage)

Case ID	#Behaviour Runs	Time Taken (in seconds)
Sale-Transaction 1	4	0.001
Sale-Transaction 2	2	< 0.001
Sale-Transaction 3	2	< 0.001
Net-Bill 1	2	< 0.001
Net-Bill 2	4	< 0.001
Secure Net-Bill	5	< 0.001

Table 7.12: Time analysis of runs concerning the expert designs category.

five false positives and two faults that we were unable to categorise. For example, in 'Sale Transaction 2', there were two wrong-ordering problems raised (see Table 7.11) that were categorised as unknown. As Figure 7.13 shows, the participant designed both messages—'Accept_The_Price' and 'Make_Payment_By_Card'—to be sent by one plan. Our approach considered that these messages could be sent in any order. However, 'Accept_The_Price' needed to be sent first to correspond to the protocol. Despite this, we did not classify this as a true positive defect, because it was unclear whether the participant assumed that the ordering between these two messages was implicit.

Considering the 'Net-Bill 1' case, the design had a plan to post both 'ExitSystem' and 'Accept' messages, while the protocol had these two messages in an alternative control fragment. Subsequently, as shown in Table 7.11, the design resulted in a longer run than it should. In contrast to the 'Sale-Transaction 1' case, the designer assumed that a selection

Case ID	True Positive	False Positive	Unknown	Total
ATS	12	1	0	13
Prepare-Arrival	0	0	0	0
Handle-Baggage	2	0	0	2
Board	0	0	0	0
De-board	2	0	0	2
Prepare-Departure	0	0	0	0
Maintain-Aircraft	0	0	0	0
Service-Aircraft	0	0	0	0
Total	16	1	0	17

Table 7.13: Categorisation of defects over all iterations for the ATS case-study.

Case ID	True Positive	False Positive	Unknown	Total
ControlSendContent Scenario	1	0	1	2
RespondToOperator Scenario	0	0	0	0
OptimiseOilProduction Scenario	1	0	1	2
Optimisation Negotiation Protocol	0	0	0	0
Total	2	0	2	4

Table 7.14: Categorisation of defects over all iterations for the oil production simulation case-study.

statement would be coded into the plan, allowing only one message to be posted and, hence, the *long run* fault was false positive.

Table 7.11 demonstrates that 66.7% of the six designs provided partial coverage as per the protocol. Although the 'Net-Bill 2' design resulted in zero defects from the first iteration, it covered 50% of the 'Net-Bill' protocol. Regarding the time required to check the conformity of the correct version of the designs (i.e., the *worst-case scenario*), all of the six cases took fractions of seconds to be verified. Table 7.12 lists the time taken to check each design. The table then shows that the longest time was taken by the 'Sale-Transaction 1' case, with 0.001 seconds taken to check four runs.

Category 3: case studies — The designs of these case studies were reconstructed through reverse engineering the implementation of these case studies and using their documenta-

Case ID	#Iter	ME	TS	TL	MO	Total	Cov
ATS	14	7	0	0	6	13	50
Prepare-Arrival	1	0	0	0	0	0	100
Handle-Baggage	3	1	0	0	1	2	100
Board	1	0	0	0	0	0	100
De-board	3	2	0	0	0	2	100
Prepare-Departure	1	0	0	0	0	0	100
Maintain-Aircraft	1	0	0	0	0	0	100
Service-Aircraft	1	0	0	0	0	0	100
Total	25	10	0	0	7	17	—

Table 7.15: Summary of potential problems raised by the tool concerning the ATS case study (#Iter: number of iterations, ME: missing entity, TS: run too short, TL: run too long, MO: mismatch between entities, Cov: percentage of coverage)

Case ID	#Iter	ME	TS	TL	MO	Total	Cov
ControlSendContent Scenario	3	0	0	0	2	2	100
RespondToOperator Scenario	1	0	0	0	0	0	100
OptimiseOilProduction Scenario	3	0	0	1	1	2	100
Optimisation Negotiation Protocol	1	0	0	0	0	0	50
Total	8	0	0	1	3	4	—

Table 7.16: Summary of potential problems raised by the tool concerning the Oil Production Stimulator case study (#Iter: number of iterations, ME: missing entity, TS: run too short, TL: run too long, MO: mismatch between entities, Cov: percentage of coverage)

tions. Therefore, the designers' assumptions were partially documented by the project team. Hence, as Tables 7.13 and 7.14 demonstrate, we were able to find 18 defects in the designs (i.e., true positives) and one false positive, while we could not categorise two faults because of the lack of documented design assumptions. For example, in the 'ATS' case, the design captured a plan that posted multiple messages and these messages needed to be posted sequentially as per the protocol. This resulted in a wrong-ordering problem. However, the implementation of the system had these two messages posted sequentially, as per the protocol; hence, it is not a defect in the design.

Tables 7.15 and 7.16 show information about the potential defects found within the designs.

Case ID	#Behaviour Runs	Time Taken (in seconds)
ATS	11933397408	4798.044
Prepare-Arrival	1	< 0.001
Handle-Baggage	1	< 0.001
Board	1	< 0.001
De-board	1	< 0.001
Prepare-Departure	1	< 0.001
Maintain-Aircraft	6	0.008
Service-Aircraft	6	0.002

Table 7.17: Time analysis of runs concerning the ATS case-study.

Case ID	#Behaviour Runs	Time Taken (in seconds)
ControlSendContent Scenario	2	0.002
RespondToOperator Scenario	1	< 0.001
OptimiseOilProduction Scenario	2	0.003
Optimisation Negotiation Protocol	1	< 0.001

Table 7.18: Time analysis of runs concerning the oil production simulation case-study.

All the faults in Table 7.15 represent defects in the designs; however, as stated above, one of the six 'MO' faults in the 'ATS' case is not a defect. Similarly, two of the three 'MO' faults in Table 7.16 are not defects, as we could not categorise them as a consequence of the lack of design assumptions. With respect to the outcomes of the coverage check, all cases aside from two protocols, fully covered their point-of-reference artefacts.

Tables 7.17 and 7.18 show the time-cost for checking the conformity of all possible behaviour runs from the agent-behaviour models. The time-cost was low for each of the cases, except for the 'ATS' case. The design resulted in 11,933,397,408 runs and took over 1.3 hours. The duration for verifying such a large quantity of runs is very samll for an industrial-scale system like the 'ATS' protocol (see page 176).

Threats to Validity

The outcomes of the evaluation we conducted can be generalised. As with all experiments, there are considerable risks in terms of validity and, in this particular case, the main external threat to the validity of our experiments was the inclusion of student assignments. These assignments were of the same specifications and resulted in the provision of eight general designs out of the 15 systems that were checked in our experiments. As these designs were developed by students, the types of defects may not be representative of those made by more experienced software engineers. However, this risk did not eventuate, as the defects found in these assignments spread across all the four fault categories (see Section 7.2.2). Another threat is that we did not conduct an initial manual analysis of the agent-behaviour models to obtain an idea of how many defects were affecting these models. Therefore, we cannot judge whether there may by defects that our approach has missed.

7.3 Scalability Evaluation

The previous section has demonstrated that our approach was scaled on two industrial-scale systems; here, we examine how well the proposed approach scales as the size of the agent design grows. The complexity of our approach rests on the behaviour run extraction from the resulting plan graphs. The number of runs grows exponentially as plan graphs increase in size and its branching factor. Therefore, we generate synthetic plan graphs, systematically varying the size and the amount of parallelism up until the time taken to check the conformity of the runs—within a 'reasonable' period. We define 'reasonable' to be within 24 hours.

7.3.1 Experiment Design

We first generated synthetic plan graphs that were a combination of plans and other design entities, including goals, messages, actions and percepts. We generated a total of 21 different plan graphs using the systematic process described below. We extracted and executed the behaviour runs, recording the following measures:

i. The number of runs corresponding to a plan graph.

ii. The time that was taken to extract and execute all runs of a plan graph against the point-of-reference artefacts Petri net.

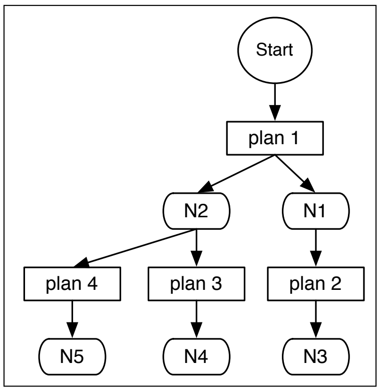


Figure 7.14: Plan graph for case 2 in Table 7.19

Plan graph generation

To increase the size of the plan graph, we increased the number of design entities, including plans, actions, percepts, goals and messages. For simplicity, rather than generating graph structures we generated *trees* (i.e., single root and acyclic trees). However, for consistency, we will continue to refer to them as plan graphs.

Plan graphs capture two control fragments: choice and parallel. It is common for plan graphs to capture both control types of fragments, so we defined the following systematic process to generate a variety of plan graphs while systematically increasing the scale of the graphs (see Figure 7.14).

1. We started all plan graphs with one plan that was linked to a number of posted entities (parallelism).
2. The type of each branch (choice or parallel) was the opposite of its parent.
3. The number of nodes N_L at level L is $N_L = N_{L-1} + c$, where N_{L-1} is the number of nodes at the parent level and c is a constant.
4. All nodes had one child, except the left-most node, which had $c+1$ children.

As is shown in Table 7.19, by varying the size of a plan graph, we vary the size of the agent design. For instance, in Case 3, the plan graph is of a breadth of two and a depth of three, which results in a design with eight plans and nine other nodes (actions, percepts, goals and messages). In Case 10, with a depth of 10, the design had 64 plans and 65 other nodes.

	B	D	#Paths	#Plans	#Nodes	#Runs	Time (in seconds)
Case 1	2	1	0	1	2	2	< 0.001
Case 2	2	2	2	4	5	12	0.001
Case 3	2	3	4	8	9	60	0.002
Case 4	2	4	6	13	14	280	0.014
Case 5	2	5	8	19	20	1260	0.039
Case 6	2	6	10	26	27	5544	0.026
Case 7	2	7	12	34	35	24024	0.207
Case 8	2	8	14	43	44	102960	0.574
Case 9	2	9	16	53	54	437580	2.325
Case 10	2	10	18	64	65	1847560	4.130
Case 11	3	1	0	1	3	6	< 0.001
Case 12	3	2	3	6	8	270	0.010
Case 13	3	3	6	13	15	8400	0.140
Case 14	3	4	9	22	24	242550	1.992
Case 15	3	5	12	33	35	6810804	37.560
Case 16	3	6	15	46	48	188684496	1156.001
Case 17	3	7	18	61	63	5187948480	61154.961
Case 18	4	1	0	1	4	24	0.001
Case 19	4	2	4	8	11	10080	0.130
Case 20	4	3	8	18	21	2587200	13.880
Case 21	4	4	12	31	34	630630000	4697.720

Table 7.19: Structures of the plan graphs used in the experiments (B: breadth, D: Depth, #Runs: number of runs generated, conformity check: time taken in seconds to extract and execute all behaviour runs)

We incremented the breadth and depth by one for each case, omitting cases for which the execution took more than one day (e.g., breadth of three , depth of eight and breadth of four, depth of five).

Execution environment

We ran all the experiments on a desktop using a 64-bit Intel Core i7 processor clocked at 3.4 GHz. One GB of RAM was dedicated for use by the Java Virtual Machine. We ran no other tasks on the machine. We ran each case six times and recorded the average times for checking the conformity of all runs.

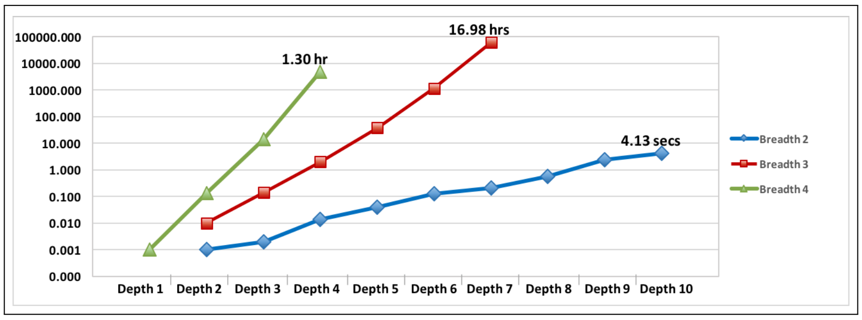


Figure 7.15: Time taken to check the conformity of runs for all the cases in Table 7.19

7.3.2 Results

Table 7.19 shows our findings. We see that the number of behaviour runs grows exponentially as the plan graph increases in size, as expected. Similarly, we see that the execution time is proportional to the number of runs.

Figure 7.15 plots the time taken to check the conformity of all the cases listed in Table 7.19. Note that the Y-axis is logarithmic. The exponential nature of this problem is hardly surprising. What we consider significant is the number of runs that can be analysed within a reasonable time frame (in our case, within one day). For instance, the time taken to extract and execute over five billion runs in Case17 (the largest case) took around 17 hours. However, as Table 7.19 shows, the equivalent design to the plan graph of case 17 consists of 61 plans and 63 other nodes (actions, percepts, goals and messages). Further, these nodes include both parallelism and choice decompositions.

While the plan graphs analysed were generated artificially, we believe that this shows that our approach can, at least under some circumstances, scale up relatively well. It is, of course, impossible to know what sizes will occur in practice without a real design of some significant size. Based on the experience within our research group over the last 15 years of building agent systems in collaboration with industry partners, Case 15 (with 33 plans and 35 nodes) is reasonably large and our approach took just 38 seconds to check it.

Threats to Validity

The main threat to external validity in our scalability analysis is the length of the entity labels from the plan graphs used in the experiments. In our approach, the conformity check is conducted via *string comparisons* and, hence, comparing labels with lengthy strings takes more time than considering short ones. The entity labels in our analysis are of a fixed length (of three). Thus, further experimentations are required on plan graphs with variable lengths of node labels for more generalised results. Another threat is that the plan graphs used are synthetic. However, as discussed above, the size of some of the cases included in the analysis are even larger than some industry-level projects (e.g., in Case 17).

7.4 Discussion

The method presented in this thesis compares the consistency of design artefacts to flag potential defects. However, given the partial nature of the agent-behaviour models used in Prometheus and other semi-formal agent-oriented software engineering methodologies, neither soundness nor completeness are possible.

Specifically, our method may raise false positives as a result of the under-specification of the designs. Given an agent plan that posts several entities, our method assumes that the plan posts all entities. However, a designer may intend only some of these entities to be posted for any single execution of the plan, depending on some logic internal to the plan. In Prometheus, such logic is not captured at the design level. As such, the set of possible runs generated by our method could be larger than the set of runs intended by the designer and some of the additional runs may violate the corresponding point-of-reference artefact.

With regard to the categorisation in Table 3.1 (page 73), some causes can result in false positives, but others will not. Causes 1 and 3(a) (short run and missing entities) will always be true positives, because the partial nature of the designs will result in more runs than may have been intended, but never fewer runs. Additional runs resulted from designs in which plans posted multiple entities, but the designer intended only some of these to be posted at any time. Our method assumes that all entities must be posted and that this will not result in shorter runs or missing entities. Conversely, Causes 2 and 3(b) in Table 3.1 may result from under-specification and therefore may contain false positives. To avoid false positives, agent-behaviour models can be structured such that a plan intended to post only a subset of its specified entities is broken into sub-plans, in which all entities specific are intended to be posted, thus providing a deterministic way of calculating the entities that will be sent by each plan.

The exponential nature of the approach raises concerns about its applicability in practice. To overcome this exponentiation challenge, our approach checks one token at a time, rather than checking the entire set of runs at once. As a result, the time to traverse the reachability graph is reduced, as the sub-graphs under the erroneous states are eliminated. However, in the *worst-case* scenario (i.e., a valid design), the entire reachability graph must be traversed. As per the scalability analysis we conducted in Section 7.3, the computation time is not prohibitively expensive in regard to the worst-case scenario. Further, we have found from the experience of building agent systems in collaboration with industry partners within our research group that designs are not large enough to take an 'unreasonable' amount of time to be verified. Thus, we believe our approach can provide relatively *quick* feedback when checking agent designs.

On the evaluation of design coverage, our approach does not report lack of coverage as a defect in the design. This is because the coverage is a designer choice. For example, a designer may choose to implement the specified point-of-reference artefact partially and still implement it correctly. Given this, we cannot say that the design is defective, as certain parts of the point-of-reference artefact are relevant to the context of the system-to-be and are functional and accurate. It is important to note that our coverage check is not a simple static check (i.e., checking whether the design entities exist in the agent-behaviour models or not). Considering the same 'Net-Bill' protocol, if we added another 'Request' message at the end of the protocol, the static check would not give any warning as to whether the design covers the OPT control fragment or not.

Finally, the verification approach in this thesis is grounded in the Prometheus methodology, but it can be generalised to other methodologies. Our work is a way of transforming the requirement specifications of Prometheus into executable structures. It is difficult to provide useful verification techniques without grounding them in a methodology that is also able to be implemented as usable tool support. We believe that adapting our work to other methodologies is possible because the most widely used agent-oriented software-engineering methodologies share similar design concepts (see [Winikoff and Padgham 2013]). Note that we use Petri nets as an intermediate representation and that any notation translatable to Petri nets will work.

CHAPTER 8

Conclusion

Multi-agent systems are gaining popularity for building complex applications ranging from critical systems used in crisis management, to non-critical systems such as video games. One of the barriers to the widespread adoption of agent technology in the industry is its reliability. To increase the trust in such systems, many testing and debugging techniques have been proposed.

It is well accepted in software engineering that defects found and fixed late in a project cost considerably more than those found earlier. Thus, the ability to find defects in the design artefacts (i.e., before the implementation phase) of the system-to-be will positively impact the cost of developing multi-agent systems. However, comparatively little research has been conducted into developing methods for detecting defects in agent designs (see Section 2.9 in Chapter 2).

In this thesis, we have proposed an approach and tool support for finding defects in agent designs concerning two point-of-reference artefacts: (1) interaction protocols and (2) scenarios. This approach involves generating the set of possible behaviour runs permitted by the agent-behaviour models and checking whether the sequences of design entities in these runs are valid with respect to the specified point-of-reference artefact. Although our tool supports only designs that have been written using the Prometheus methodology, as discussed in Section 7.4, we believe that the approach is general enough to work with other AOSE methodologies that follow the BDI model of agency. In this thesis, we have addressed the following research aims:

First, we determined the design units to be considered in our approach. As is stated in the conceptual framework of our approach (see Chapter 3), we identified the point-of-reference artefacts and the design units considered for the proposed approach. Our approach considers two point-of-reference artefacts: (1) interaction protocols and (2) scenarios. Regarding the agent-behaviour models, the verification framework considers all the design entities that are used to specify agents in Prometheus. However, we excluded data-sets, unless they performed actions (e.g., posting an internal event that triggered plans).

Second, we showed how the point-of-reference artefacts of the system-to-be were transformed to executable models. A scenario in Prometheus is a sequence of steps and these steps can be of different types, including goals. Thus, a scenario may have alternative paths, produced through consideration of information from the goal model relevant to that scenario. Also, Prometheus does not support structured specification for variations and, hence, it is tedious to include the variations in the checking process.

We overcome the limitations mentioned above by automatically generating an activity diagram from a given scenario and the goal model of the system-to-be. The activity diagram then acted as a coherent structure for requirements and permitted the software engineers to specify variations as part of a more structured approach that allowed reasoning about such variations. The details of this construction process along with the user study we conducted were explained in Chapter 4.

After generating the activity diagram, we transformed the point-of-reference artefacts in Prometheus: (1) the UML-activity diagrams (representing requirements) and (2) the AUML-sequence diagrams (representing interaction protocols) into *place/transition* Petri nets. We chose Petri nets over other formalisms, such as statecharts, since they are simple to understand and they serve our purpose. Additionally, there are existing algorithms proposed by Poutakidis et al. [Padgham et al. 2005b] that translate AUML-interaction fragments (i.e., sequential, selection, loops and parallelism control fragments) into Petri nets. In fact, we adopted the work of Poutakidis et al. [Padgham et al. 2005b] with some modifications as detailed in Chapter 5.

Third, we proposed a technique that permits the extraction of all possible runs from the agent-behaviour models concerning the specified point-of-reference artefact. This includes transforming agent-behaviour models into executable models. As explained in Chapter 5, we adopted the concept of a *plan graph* proposed by Miller et al. [Miller et al. 2010], to construct a coherent structure that merges different agent-behaviour models related to the specified point-of-reference artefact.

Then, we transformed the plan graph into a Petri net to allow for the extraction of all possible behaviour runs via the Petri net's reachability graph.

Fourth, we ensured the conformity of agent-behaviour models with the specified point-of-reference artefact by comparing the two Petri nets (i.e., the Petri nets of the agent-behaviour models and the point-of-reference artefact of the system-to-be). Such comparisons were performed by extracting all possible runs from the agent-behaviour models and checking them against the point-of-reference artefact's Petri net. As per Chapter 6, a correctness check was performed by executing the possible behavioural runs extracted from the plan graph against the point-of-reference artefact's Petri net and logging the existence of any violations. Hence, the output of our verification framework is a report that documents information about the agent design under investigation. We implemented this verification approach as an Eclipse plug-in that integrated with PDT.

To show the effectiveness of the verification framework in this thesis, we evaluated it on 11 versions of the same design (via mutations) that were carried out by two participants. The results demonstrated that our approach was able to detect defects successfully.

Next, we evaluated our approach in terms of 15 implemented systems; the total number of point-of-reference artefacts in these systems was 32. As detailed in Chapter 7, we categorised these cases into three categories: (1) students assignments, (2) expert designs and (3) case studies. Overall, the approach detected 100 faults in total, 89% of which were defects in the designs (i.e., true positives). Because of the lack of documented assumptions about the designs, 6% of the 100 faults were false positives and 5% we could not categorise (i.e., they remained unknown). The 89% of the defects were distributed across the four faults categories (wrong ordering, missing entity, run too short and run too long). This shows that our verification framework is able to detect defects in agent designs with a low number of false positives and generally in a negligible amount of time.

Additionally, we performed a scalability analysis to examine how well the proposed approach scales as the size of the agent design grows. In this analysis, we synthesised plan graphs, systematically varying the size and the amount of parallelism. The largest plan graph used in our analysis was of a depth of seven and a breadth of three. The plan graph contained 61 plans and 63 other design entities (i.e., goals, percepts, actions and events). The plan graph resulted in more than five billion runs and the time taken for checking their conformity was 16.98 hours. The results of our scalability analysis demonstrate that our approach could verify significant plan graphs in under 24 hours, despite the exponential explosion in runs as the designs grow.

Limitations

As with any research project, there are limitations that require further investigations. In this thesis, the verification approach considers one point-of-reference artefact at a time. Therefore, our approach is unable to detect defects that result from conflicts between multiple point-of-reference artefacts. In other words, our approach is a *unit verification* approach rather than an *integration verification* approach. The integration verification would detect inconsistencies in the agent-behaviour models with respect to multiple point-of-reference artefacts.

Another limitation of our proposed method is the false positives that may be raised as a result of the lack of design assumptions. Even though our approach did not result in a high number of false positives, the approach needs to be refined to reduce these false positives to a minimum.

Further, the approach lacks a detailed fault model that categorises design defects based on their severity. Our approach reports all defects detected as errors in the design, whereas some defects may be treated as warnings by some designers. For example, in our approach, a behaviour run that has more tokens than it should is marked as an erroneous run and reflects an error in the design. However, based on the context of the system-to-be, such erroneous runs can be interpreted as warnings.

Our evaluation indicated that some designs could result in an explosive number of runs and, therefore, in long execution times. This exponential nature of the approach represents another limitation that could be addressed.

Regarding the evaluation results, there were some threats to their generalisability. The main threat to the validity of our evaluation was the inclusion of student assignments. Since these systems were designed by relativity inexperienced participants, the defects detected may not be representative of those made by more experienced participants.

Future Work

There are many potential extensions of this topic. Future work might address some of the limitations.

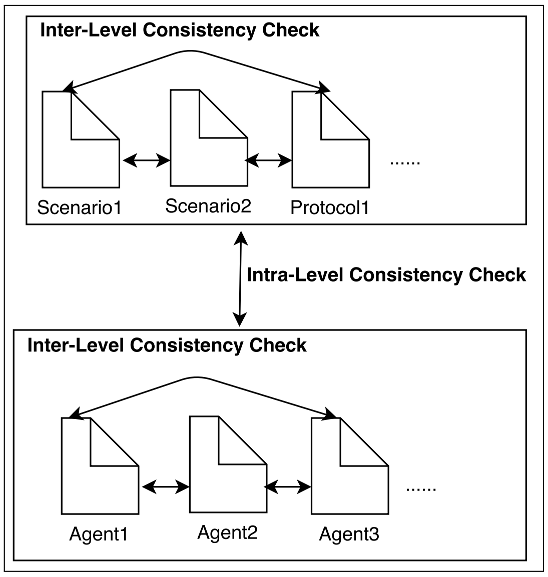

Inter-Level Consistency Check
Scenario1
Scenario2
Protocol1
......
Intra-Level Consistency Check
Inter-Level Consistency Check
Agent1
Agent2
Agent3
......

Figure 8.1: Checking the operational consistency of the system-to-be

- Our approach considers one point-of-reference artefact at a time. Thus, a natural extension that fits the context of this research project is the consideration of the overall picture—that is, of all the scenarios and protocols at once (i.e., integration verification). This integration verification would allow software engineers to ensure that there would be no conflicts between agents at run-time (i.e., *operational consistency*). Such a consistency check could be performed via an inter-level check and an intra-level check (see Figure 8.1). The inter-level check is an initial check of the point-of-reference artefacts and the agent-behaviour models. Software engineers could test a number of properties, such as operational conflicts, in the point-of-reference artefacts [Thangarajah and Padgham 2011; 2004, Thangarajah et al. 2003; 2002]. Similarly, designers may examine the agent-behaviour models to ensure that they are conflict free [Shapiro et al. 2012].

- A refinement of our approach is needed to reduce the number of false positives. False positives result because the design of the system-to-be lacks details, such as assumptions on parallelisms. One way to overcome this limitation would be by extending Prometheus such that it allows designers to provide more details on their designs and include such data in the vitrification process; for example, to include context conditions of the plans in the agent-behaviour models. These conditions could then be used to filter out some runs that currently lead to false positives. Another approach would be to extend the tool-support. Such an extension may include offering of a pop-up dialogue box to prompt software engineers to state their assumptions about the uncertainties in the design under consideration; for example, to prompt for assumptions about parallelisms in the design.

The fault model in our approach is limited to a certain extent. Our approach does not categorise defects based on their severity. In other words, our approach reports all defects as errors, whereas in some cases some defects, based on the designers' interpretations, can be treated as warnings. Therefore, there is a need to enhance the fault model of our approach by providing a broader categorisation of defects. To develop such an enhanced fault model, a basic user testing of our verification framework is required, with a number of software engineering experts involved.

- To improve the scalability of our approach, the depth-first-search algorithm used to extract behaviour runs needs to be enhanced. Currently, the search algorithm extracts the first path of the graph and follows it until it reaches the last path; the algorithm moves in the order in which the paths appear. A *prioritisation* mechanism, via some sort of heuristics, could be applied to the extraction algorithm to mitigate the exponential nature of our approach. The goal of this prioritisation mechanism would be to find and report defects early; hence, the erroneous parts of the reachability graph under consideration would be eliminated early in the checking process.
- In terms of activity diagrams, further investigation is needed into ways to provide *round-trip engineering*. This could be achieved by designing and implementing an algorithm to automate the process for propagating changes from the activity diagram into the goal model and possibly into the scenario.

Final Comment

This thesis contributes a framework and a methodology that achieves the goal of identifying defects early in the development life-cycle for BDI systems. The basic intuition of our verification framework is to extract the possible behaviour runs from the agent-behaviour models and to verify whether these runs conform to the specifications of the system-to-be. We used Petri nets to act as executable models for both the agent-behaviour models and the point-of-reference artefacts of the system-to-be. The tool-support of our verification framework accepts a Prometheus-based design file and produces a report. The report logs the defects in the agent-behaviour models concerning the specified point-of-reference artefact (i.e., its requirements or protocol) in the design file. The evaluation we conducted showed that the verification framework is effective for detecting defects in BDI systems.

Bibliography

Y. Abushark and J. Thangarajah. Propagating AUML protocols to detailed design. In M. Cossentino, A. E. Fallah-Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems - First International Workshop, EMAS 2013, St. Paul, MN, USA, May 6-7, 2013, Revised Selected Papers*, volume 8245 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2013. doi: 10.1007/978-3-642-45343-4_2.

S. Aknine, S. Pinson, and M. F. Shakun. An extended multi-agent negotiation protocol. *Autonomous Agents and Multi-Agent Systems*, 8(1):5–45, 2004.

E. Al-Hashel, B. M. Balachandran, and D. Sharma. A comparison of three agent-oriented software development methodologies: Roadmap, prometheus, and mase. In B. Apolloni, R. J. Howlett, and L. C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems, 11th International Conference, KES 2007, XVII Italian Workshop on Neural Networks, Vietri sul Mare, Italy, September 12-14, 2007, Proceedings, Part III*, volume 4694 of *Lecture Notes in Computer Science*, pages 909–916. Springer, 2007. doi: 10.1007/978-3-540-74829-8_111.

M. N. Alam, S. Hossain, and K. N. E. Alam. Use case application in requirements analysis using secure tropos to UMLsec-security issues. *International Journal of Computer Applications*, 109 (4):21–25, 2015.

P. Anderson, T. W. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, 2003. doi: 10.1109/MS.2003.1207453.

J. L. Arcos, M. Esteva, P. Noriega, J. A. Rodríguez-Aguilar, and C. Sierra. An integrated development environment for electronic institutions. In *Software agent-based applications, platforms and development kits*, pages 121–142. Springer, 2005.

A. Artikis, M. Sergot, and J. Pitt. An executable specification of a formal argumentation protocol. *Artificial Intelligence*, 171(10):776–804, 2007.

M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Verification of protocol conformance and agent interoperability. In *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI, London, UK, June 27-29, 2005, Revised Selected and Invited Papers*, pages 265–283, 2005.

M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. A priori conformance verification for guaranteeing interoperability in open environments. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, pages 339–351, 2006.

M. Baldoni, C. Baroglio, A. K. Chopra, N. Desai, V. Patti, and M. P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, pages 843–850, 2009.

M. Baldoni, C. Baroglio, and E. Marengo. Commitment-based protocols with behavioral rules and correctness properties of MAS. In *Declarative Agent Languages and Technologies VIII - 8th International Workshop, DALT 2010, Toronto, Canada, May 10, 2010, Revised, Selected and Invited Papers*, pages 60–77, 2010.

M. Barbuceanu and M. S. Fox. COOL: A language for describing coordination in multi agent systems. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 17–24. The MIT Press, 1995.

B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.

F. Bergenti and A. Poggi. Exploiting UML in the design of multi-agent systems. In *Engineering Societies in the Agents World*, pages 106–113. Springer, 2000.

G. Beydoun, G. C. Low, B. Henderson-Sellers, H. Mouratidis, J. J. Gómez-Sanz, J. Pavón, and C. Gonzalez-Perez. FAML: A generic metamodel for MAS development. *IEEE Trans. Software Eng.*, 35(6):841–863, 2009. doi: 10.1109/TSE.2009.34.

B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984. doi: 10.1109/MS.1984.233702.

B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Software Eng.*, 14(10):1462–1477, 1988. doi: 10.1109/32.6191.

B. W. Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981. ISBN 978-0138221225.

N. Bolloju and S. X. Sun. Benefits of supplementing use case narratives with activity diagramsan exploratory study. *Journal of Systems and Software*, 85(9):2182–2191, 2012.

R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*, pages 409–416. ACM, 2003. doi: 10.1145/860575.860641.

R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006a.

R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006b. doi: 10.1007/s10458-006-5955-7.

R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007. ISBN 978-0470029008.

T. Bosse, D. N. Lam, and K. S. Barber. Automated analysis and verification of agent behavior. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*, pages 1317–1319. ACM, 2006. doi: 10.1145/1160633.1160876.

J. A. Botía, A. López-Acosta, and A. G. Skarmeta. ACLAnalyser: a tool for debugging multi-agent system. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 967–968. IOS Press, 2004.

J. A. Botía, J. J. Gómez-Sanz, and J. Pavón. Intelligent data analysis for the verification of multi-agent systems interactions. In E. Corchado, H. Yin, V. J. Botti, and C. Fyfe, editors, *Intelligent Data Engineering and Automated Learning - IDEAL 2006, 7th International Conference, Burgos, Spain, September 20-23, 2006, Proceedings*, volume 4224 of *Lecture Notes in Computer Science*, pages 1207–1214. Springer, 2006. doi: 10.1007/11875581_143.

P. Bourque, R. E. Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014.

M. Bratman. *Intention, plans, and practical reason.* Harvard University Press, 1987.

M. Bratman. *Faces of intention: Selected essays on intention and agency.* Cambridge University Press, 1999.

M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.

L. Bratthall and C. Wohlin. Is it possible to decorate graphical software design and architecture models with qualitative information?-an experiment. *IEEE Trans. Software Eng.*, 28(12):1181–1193, 2002. doi: 10.1109/TSE.2002.1158290.

P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in Tropos: A transformation based approach. In M. Wooldridge, G. Weiß, and P. Ciancarini, editors, *Agent-Oriented Software Engineering II, Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001, Revised Papers and Invited Contributions*, volume 2222 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2001. doi: 10.1007/3-540-70657-7_11.

P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004. doi: 10.1023/B:AGNT.0000018806.20944.ef.

P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents-components for intelligent agents in java. *AgentLink News Letter*, 2(1):2–5, 1999.

M. W. Bush. Improving software quality: The use of formal inspections at the JPL (experience report). In F. Valette, P. A. Freeman, and M. Gaudel, editors, *Proceedings of the 12th International Conference on Software Engineering, Nice, France, March 26-30, 1990.*, pages 196–199. IEEE Computer Society, 1990.

A. Castor, R. C. Pinto, C. T. L. L. Silva, and J. Castro. Towards requirement traceability in Tropos. In M. Ridao and L. M. Cysneiros, editors, *Anais do WER04 - Workshop em Engenharia de Requisitos, Tandil, Argentina, Dezembro 9-10, 2004*, pages 189–200, 2004.

J. Castro, R. C. Pinto, A. Castor, and J. Mylopoulos. Requirements traceability in agent oriented development. In A. F. Garcia, C. J. P. de Lucena, F. Zambonelli, A. Omicini, and J. Castro, editors, *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications [the book is a result of SELMAS 2002]*, volume 2603 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2002. doi: 10.1007/3-540-35828-5_4.

L. Cernuzzi and F. Zambonelli. Experiencing AUML in the GAIA methodology. In *ICEIS 2004, Proceedings of the 6th International Conference on Enterprise Information Systems, Porto, Portugal, April 14-17, 2004*, pages 283–288, 2004.

L. Cernuzzi and F. Zambonelli. Gaia4e: A tool supporting the design of MAS using gaia. In J. Cordeiro and J. Filipe, editors, *ICEIS 2009 - Proceedings of the 11th International Conference on Enterprise Information Systems, Volume SAIC, Milan, Italy, May 6-10, 2009*, pages 82–88, 2009.

A. Chella, M. Cossentino, and L. Sabatucci. Tools and patterns in designing multi-agent systems with PASSI. *WSEAS Transactions on Communications*, 3(1):352–358, 2004.

E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-oriented development: the fusion method*. Prentice-Hall, Inc., 1994.

L. C. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In T. Chen, D. N. Serpanos, and W. Taha, editors, *International Conference on Embedded Software and Systems, ICESS '09, Hangzhou, Zhejiang, P. R. China, May 25-27, 2009.*, pages 396–403. IEEE Computer Society, 2009.

M. Cossentino. From requirements to code with the PASSI methodology. *Agent-oriented methodologies*, 3690:79–106, 2005.

R. S. Cost, Y. Chen, T. W. Finin, Y. Labrou, and Y. Peng. Using colored petri nets for conversation modeling. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2000. doi: 10.1007/10722777_12.

B. Cox. Netbill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce, New York, New York, USA, July 11-12, 1995*, pages 77–88. USENIX Association, 1995.

H. K. Dam and M. Winikoff. Towards a next-generation AOSE methodology. *Sci. Comput. Program.*, 78(6):684–694, 2013. doi: 10.1016/j.scico.2011.12.005.

M. Dastani, J. Brandsema, A. Dubel, and J. C. Meyer. Debugging BDI-based multi-agent programs. In L. Braubach, J. Briot, and J. Thangarajah, editors, *Programming Multi-Agent Systems - 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers*, volume 5919 of *Lecture Notes in Computer Science*, pages 151–169. Springer, 2009. doi: 10.1007/978-3-642-14843-9_10.

M. Dastani, K. V. Hindriks, and J.-J. Meyer. *Specification and verification of multi-agent systems*. Springer Science & Business Media, 2010. ISBN 978-1-4419-6984-2.

E. David, R. Azoulay-Schwartz, and S. Kraus. An english auction protocol for multi-attribute items. In *Agent-Mediated Electronic Commerce IV. Designing Mechanisms and Systems*, pages 52–68. Springer, 2002.

S. Deloach. The MaSE methodology. *Methodologies and software engineering for agent systems*, pages 107–125, 2004.

S. DeLoach, L. Padgham, A. Perini, and A. Susi. Using three aose toolkits to develop a sample design. *International Journal of Agent-Oriented Software Engineering IJAOSE*, 3(4):416–476, 2009. doi: 10.1504/IJAOSE.2009.025321.

S. A. DeLoach. Multiagent systems engineering: a methodology and language for designing agent systems. Technical report, DTIC Document, 1999.

S. A. DeLoach. Analysis and design using MaSE and agentTool. Technical report, DTIC Document, 2001.

S. A. DeLoach and J. C. Garcia-Ojeda. O-MaSE: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001. doi: 10.1142/S0218194001000542.

L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Autom. Softw. Eng.*, 19(1):5–63, 2012. doi: 10.1007/s10515-011-0088-x.

J. Desel and W. Reisig. Place/transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, pages 122–173. Springer, 1998.

V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008. doi: 10.1109/TCAD.2008.923410.

C. Duran-Faundez, M. Ramos, and P. Rodriguez. Applying gaia and AUML for the development of multiagent-based control software for flexible manufacturing systems: addressing methodological and implementation issues. *Softw., Pract. Exper.*, 45(12):1719–1737, 2015. doi: 10.1002/spe.2302.

L. Engmo and L. Hallen. Software agents applied in oil production. Technical report, Institutt for datateknikk og informasjonsvitenskap, 2007.

M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 1045–1052, 2002.

R. Evertsz, J. Thangarajah, N. Yadav, and T. Li. Tactics development framework (demonstration). In A. L. C. Bazzan, M. N. Huhns, A. Lomuscio, and P. Scerri, editors, *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, pages 1639–1640. IFAAMAS/ACM, 2014.

R. Evertsz, J. Thangarajah, N. Yadav, and T. Ly. A framework for modelling tactical decision-making in autonomous systems. *Journal of Systems and Software*, 110:222–238, 2015. doi: 10.1016/j.jss.2015.08.046.

M. E. Fagan. Design and code inspections to reduce errors in program development. In *Pioneers and Their Contributions to Software Engineering*, pages 301–334. Springer, 2001.

G. A. C. Filho and A. Zisman. Traceability and completeness checking for agent-oriented systems. In R. L. Wainwright and H. Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 71–77. ACM, 2008. doi: 10.1145/1363686.1363706.

T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

A. Fipa. Message structure specification, 2002.

A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *Proceedings of 5th IEEE International Symposium on Requirements Engineering*, pages 174–181. IEEE, 2001.

I. García-Magariño, J. J. Gómez-Sanz, and R. Fuentes-Fernández. Model transformations for improving multi-agent system development in INGENIAS. In M. P. Gleizes and J. J. Gómez-Sanz, editors, *Agent-Oriented Software Engineering X - 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers*, volume 6038 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2009. doi: 10.1007/978-3-642-19208-1_4.

C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering (2. ed.)*. Prentice Hall, 2003. ISBN 978-0-13-305699-0.

J. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney, and G. Memmi. Scrutiny: A collaborative inspection and review system. In I. Sommerville and M. Paul, editors, *Software Engineering - ESEC '93, 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany, September 13-17, 1993, Proceedings*, volume 717 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 1993. doi: 10.1007/3-540-57209-0_24.

P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Eng. App. of AI*, 18(2):159–171, 2005.

J. J. Gómez-Sanz and R. Fuentes-Fernández. Understanding agent-oriented software engineering methodologies. *Knowledge Eng. Review*, 30(4):375–393, 2015. doi: 10.1017/S0269888915000053.

J. J. Gómez-Sanz and J. Pavón. Implementing multi-agent systems organizations with INGENIAS. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems, Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, volume 3862 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 2005. doi: 10.1007/11678823_15.

J. J. Gómez-Sanz, R. Fuentes, J. Pavón, and I. García-Magariño. INGENIAS development kit: a visual multi-agent system development environment. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Demo Proceedings*, pages 1675–1676. IFAAMAS, 2008. doi: 10.1145/1402744.1402760.

A. Gross and J. Dörr. EPC vs. UML activity diagram - two experiments examining their usefulness for requirements engineering. In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*, pages 47–56. IEEE Computer Society, 2009. doi: 10.1109/RE.2009.30.

J. J. Gutiérrez, C. Nebut, M. J. Escalona, M. Mejías, and I. M. Ramos. Visualization of use cases through automatically generated activity diagrams. In *Model Driven Engineering Languages and Systems*, pages 83–96. Springer, 2008.

D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3): 231–274, 1987. doi: 10.1016/0167-6423(87)90035-9.

M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013. ISBN 978-0-470-38737-5.

G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi: 10.1109/32.588521.

M.-P. Huget and J. Odell. Representing agent interaction protocols with agent UML. In *Agent-Oriented Software Engineering V*, pages 16–30. Springer, 2004.

IEEE. *IEEE Standard for System and Software Verification and Validation*, May 2012.

D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. doi: 10.1145/505145.505149.

I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley, 1992. ISBN 978-0-201-54435-0.

G. Jayatilleke. *A model driven component agent framework for domain experts*. PhD thesis, School of Computer Science & IT, RMIT University, 2007.

G. B. Jayatilleke, J. Thangarajah, L. Padgham, and M. Winikoff. Component agent framework for domain-experts (cafne) toolkit. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *5th International Joint Conference on Autonomous Agents and Multiagent*

Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006, pages 1465–1466. ACM, 2006. doi: 10.1145/1160633.1160917.

N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001. doi: 10.1145/367211.367250.

K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.

T. Juan, A. R. Pearce, and L. Sterling. ROADMAP: extending the gaia methodology for complex open systems. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 3–10. ACM, 2002. doi: 10.1145/544741.544744.

J. Jürjens. UMLsec: Extending UML for secure systems development. In *ł UML2002The Unified Modeling Language*, pages 412–425. Springer, 2002.

N. Kececi, W. A. Halang, and A. Abran. A semi-formal method to verify correctness of functional requirements specifications of complex systems. In B. Kleinjohann, K. H. Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems, IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002), August 25-29, 2002, Montréal, Québec, Canada*, volume 219 of *IFIP Conference Proceedings*, pages 61–69. Kluwer, 2002.

D. Kinny, M. P. Georgeff, and A. S. Rao. A methodology and modelling technique for systems of BDI agents. In W. V. de Velde and J. W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1996. doi: 10.1007/BFb0031846.

M. Köhler, D. Moldt, and H. Rölke. Modelling the structure and behaviour of Petri net agents. In *Applications and Theory of Petri Nets 2001*, pages 224–241. Springer, 2001.

J. M. A. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 2010.

T. Lacey and S. A. DeLoach. *Automatic verification of multiagent conversations*. Defense Technical Information Center, 2000.

D. N. Lam and K. S. Barber. Comprehending agent software. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 586–593. ACM, 2005. doi: 10.1145/1082473.1082562.

A. A. Lopez-Lorca, G. Beydoun, R. Valencia-García, and R. Martínez-Béjar. Supporting agent oriented requirement analysis with ontologies. *Int. J. Hum.-Comput. Stud.*, 87:20–37, 2016. doi: 10.1016/j.ijhcs.2015.10.007.

F. Macdona, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. In *Seventh International Workshop on Computer-Aided Software Engineering, 1995. Proceedings.*, pages 340–349. IEEE, 1995.

F. MacDonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Automating the software inspection process. *Autom. Softw. Eng.*, 3(3/4):193–218, 1996. doi: 10.1007/BF00132566.

V. Mashayekhi, J. M. Drake, W. Tsai, and J. Riedl. Distributed, collaborative software inspection. *IEEE Software*, 10(5):66–75, 1993. doi: 10.1109/52.232404.

H. Mazouzi, A. E. Fallah-Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 517–526. ACM, 2002. doi: 10.1145/544862.544866.

S. J. Mellor. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

T. Miller, L. Padgham, and J. Thangarajah. Test coverage criteria for agent interaction testing. In D. Weyns and M. P. Gleizes, editors, *Agent-Oriented Software Engineering XI - 11th International Workshop, AOSE 2010, Toronto, Canada, May 10-11, 2010, Revised Selected Papers*, volume 6788 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2010. doi: 10.1007/978-3-642-22636-6_6.

T. Miller, B. Lu, L. Sterling, G. Beydoun, and K. Taveter. Requirements elicitation and specification using the agent paradigm: The case study of an aircraft turnaround simulator. *IEEE Trans. Software Eng.*, 40(10):1007–1024, 2014. doi: 10.1109/TSE.2014.2339827.

S. Misra, V. Kumar, and U. Kumar. Goal-oriented or scenario-based requirements engineering technique-what should a practitioner select? In *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 2288–2292. IEEE, 2005.

N. Mitakides, P. Delias, and N. I. Spanoudakis. Validating requirements using gaia roles models. In M. Baldoni, L. Baresi, and M. Dastani, editors, *Engineering Multi-Agent Systems - Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers*, volume 9318 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2015. doi: 10.1007/978-3-319-26184-3_10.

M. Morandini, D. C. Nguyen, A. Perini, A. Siena, and A. Susi. Tool-supported development with Tropos: The conference management system case study. In *Agent-Oriented Software Engineering VIII*, pages 182–196. Springer, 2007.

S. Munroe, T. Miller, R. Belecheanu, M. Pechoucek, P. McBurney, and M. Luck. Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. *Knowledge Eng. Review*, 21(4):345–392, 2006. doi: 10.1017/S0269888906001020.

Y. Murakami, K. Minami, T. Kawasoe, and T. Ishida. Multi-agent simulation for crisis management. In *IEEE Workshop on Knowledge Media Networking, 2002. Proceedings.*, pages 135–139. IEEE, 2002.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4): 541–580, 1989.

C. D. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. *Autonomous Agents and Multi-Agent Systems*, 25(2):260–283, 2012. doi: 10.1007/s10458-011-9175-4.

D. C. Nguyen, A. Perini, and P. Tonella. A goal-oriented software testing methodology. In *Agent-Oriented Software Engineering VIII*, pages 58–72. Springer, 2007.

D. C. Nguyen, A. Perini, and P. Tonella. eCAT: a tool for automating test cases generation and execution in testing multi-agent systems. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Demo Proceedings*, pages 1669–1670. IFAAMAS, 2008. doi: 10.1145/1402744.1402757.

I. Object Management Group. *OMG Unified Modelling Language version 2.4 (OMG UML), Superstructure*. Object Management Group, 109 Highland Ave, Needham, MA 02494 USA, 2011.

J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. *Ann Arbor*, 1001:48–103, 2000.

J. Odell, H. Parunak, and B. Bauer. Agent UML: A formalism for specifying multi-agent interactions. In *Agent-Oriented Software Engineering*, pages 91–103, 2001a.

J. J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Agent-oriented software engineering*, pages 121–140. Springer, 2001b.

L. Padgham and M. Winikoff. *Developing intelligent agent systems: A practical guide*, volume 13. John Wiley & Sons, 2005.

L. Padgham, J. Thangarajah, and M. Winikoff. Tool support for agent development using the prometheus methodology. In *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*, pages 383–388. IEEE Computer Society, 2005a. doi: 10.1109/QSIC.2005.66.

L. Padgham, M. Winikoff, and D. Poutakidis. Adding debugging support to the prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173–190, 2005b.

L. Padgham, J. Thangarajah, and M. Winikoff. The prometheus design tool–a conference management system case study. In *Agent-Oriented Software Engineering VIII*, pages 197–211. Springer, 2007.

L. Padgham, J. Thangarajah, and M. Winikoff. Prometheus design tool. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1882–1883. AAAI Press, 2008.

L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Trans. Software Eng.*, 39(9):1230–1244, 2013. doi: 10.1109/TSE.2013.10.

N. Palmer. XML process definition language. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 3601–3601. Springer US, 2009. doi: 10.1007/978-0-387-39940-9_1550.

D. L. Parnas and M. Lawford. The role of inspection in software quality assurance. *IEEE Trans. Software Eng.*, 29(8):674–676, 2003. doi: 10.1109/TSE.2003.1223642.

S. Paurobally, J. Cunningham, and N. R. Jennings. Developing agent interaction protocols graphically and logically. In *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited Papers*, pages 149–168, 2003.

J. Pavón and J. J. Gómez-Sanz. Agent oriented software engineering with INGENIAS. In V. Marík, J. P. Müller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings*, volume 2691 of *Lecture Notes in Computer Science*, pages 394–403. Springer, 2003. doi: 10.1007/3-540-45023-8_38.

J. Pavón, J. J. Gómez-Sanz, and R. Fuentes. The INGENIAS methodology and tools. *Agent-oriented methodologies*, 9:236–276, 2005.

M. Pechoucek and V. Marík. Industrial deployment of multi-agent technologies: review and selected case studies. *Autonomous Agents and Multi-Agent Systems*, 17(3):397–431, 2008. doi: 10.1007/s10458-008-9050-0.

L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. From stakeholder intentions to software agent implementations. In E. Dubois and K. Pohl, editors, *Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006, Proceedings*, volume 4001 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2006. doi: 10.1007/11767138_31.

M. Perepletchikov and L. Padgham. Systematic incremental development of agent systems, using prometheus. In *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*, pages 413–418. IEEE Computer Society, 2005. doi: 10.1109/QSIC.2005.60.

A. Perini and A. Susi. Automating model transformations in agent-oriented modelling. In J. P. Müller and F. Zambonelli, editors, *Agent-Oriented Software Engineering VI, 6th International Workshop, AOSE 2005, Utrecht, The Netherlands, July 25, 2005. Revised and Invited Papers*, volume 3950 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 2005. doi: 10.1007/11752660_13.

J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0136619835.

A. Pokahr and L. Braubach. A survey of agent-oriented development tools. In *Multi-Agent Programming:*, pages 289–329. Springer, 2009.

A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-agent programming*, pages 149–174. Springer, 2005.

D. Poutakidis. *Debugging multi-agent systems with design documents*. PhD dissertation, RMIT Univeristy, 2008.

M. Purvis and S. Cranefield. Agent modelling with Petri nets. Technical report, University of Otago, 1996.

A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. F. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pages 473–484. Morgan Kaufmann, 1991.

A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In B. Nebel, C. Rich, and W. R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, October 25-29, 1992.*, pages 439–449. Morgan Kaufmann, 1992.

A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 312–319. The MIT Press, 1995.

P. Royston. The W-test for normality. *Applied Statistics*, pages 547–551, 1995.

J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

S. R. Schach. *Object-oriented software engineering*. McGraw-Hill, 2008.

S. Shapiro, Y. Lespérance, and H. J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 19–26. ACM, 2002. doi: 10.1145/544741.544746.

S. Shapiro, S. Sardiña, J. Thangarajah, L. Cavedon, and L. Padgham. Revising conflicting intention sets in BDI agents. In *AAMAS*, pages 1081–1088. IFAAMAS, 2012.

D. Singh and L. Padgham. Community evacuation planning for bushfires using agent-based simulation: Demonstration. In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*, pages 1903–1904. ACM, 2015.

I. Sommerville. *Software Engineering*. Pearson, 2009.

I. Sommerville and P. Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997. ISBN 978-0-471-97444-4.

A. M. Stavely. High-quality software through semiformal specification and verification. In *12th Conference on Software Engineering Education and Training, 22-24 March, 1999, New Orleans, Louisiana, USA*, pages 145–155. IEEE Computer Society, 1999. doi: 10.1109/CSEE.1999.755196.

L. Sterling and K. Taveter. *The Art of Agent-Oriented Modeling*. MIT Press, 2009.

J. Sudeikat, L. Braubach, A. Pokahr, and W. Lamersdorf. Evaluation of agent–oriented software methodologies–examination of the gap between modeling and platform. In *Agent-Oriented Software Engineering V*, pages 126–141. Springer, 2004.

J. Sudeikat, L. Braubach, A. Pokahr, W. Lamersdorf, and W. Renz. Validation of BDI agents. In *Programming Multi-Agent Systems*, pages 185–200. Springer, 2006.

J. Thangarajah and L. Padgham. An empirical evaluation of reasoning about resource conflicts. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*, pages 1298–1299. IEEE Computer Society, 2004. doi: 10.1109/AAMAS.2004.10225.

J. Thangarajah and L. Padgham. Computationally effective reasoning about goal interactions. *J. Autom. Reasoning*, 47(1):17–56, 2011.

J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In F. van Harmelen, editor, *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI 2002, Lyon, France, July 2002*, pages 18–22. IOS Press, 2002.

J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *IJCAI-03*, pages 721–726, 2003.

J. Thangarajah, G. B. Jayatilleke, and L. Padgham. Scenarios for system requirements traceability and testing. In L. Sonenberg, P. Stone, K. Tumer, and P. Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 285–292. IFAAMAS, 2011.

S. R. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In S. B. Jones and D. G. Novick, editors, *Proceedings of the 21st annual international conference on Documentation, SIGDOC 2003, San Francisco, CA, USA, October 12-15, 2003*, pages 184–191. ACM, 2003. doi: 10.1145/944868.944908.

M. Toranzo. *A Framework to Improve Requirements Traceability*. PhD thesis, Centro de Informatica daUniversidade Federal de Pernambuco, 2002.

G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In B. Hailpern, L. M. Northrop, and A. M. Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 47–56. ACM, 1999. doi: 10.1145/320384.320389.

I. Trencanský and R. Cervenka. Agent modeling language (AML): A comprehensive approach to modeling MAS. *Informatica (Slovenia)*, 29(4):391–400, 2005.

M. Usman, A. Nadeem, T.-h. Kim, and E.-s. Cho. A survey of consistency checking techniques for UML models. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 57–62. IEEE, 2008.

R. Valk. Object Petri nets. In *Lectures on Concurrency and Petri Nets*, pages 819–848. Springer, 2004.

A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 5–19. ACM, 2000. doi: 10.1145/337180.337184.

A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, pages 249–262. IEEE Computer Society, 2001. doi: 10.1109/ISRE.2001.948567.

A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009. ISBN 978-0-470-01270-3.

J. Wang. *Timed Petri nets: Theory and application*, volume 9. Springer Science & Business Media, 2012.

P. William. *Effective methods for software testing*. New York, NY: John Wiley & Sons, Inc, 2006.

M. Winikoff. Towards making agent UML practical: A textual notation and a tool. In *Fifth International Conference on Quality Software, 2005.(QSIC 2005).*, pages 401–406. IEEE, 2005a.

M. Winikoff. Jack intelligent agents: an industrial strength platform. In *Multi-Agent Programming*, pages 175–193. Springer, 2005b.

M. Winikoff and L. Padgham. Agent oriented software engineering. *Multiagent systems*, pages 695–757, 2013.

M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. *AI 2001: Advances in Artificial Intelligence*, pages 557–568, 2001.

M. Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 8-9, 1994, Proceedings*, volume 890 of *Lecture Notes in Computer Science*, pages 1–39. Springer, 1994. doi: 10.1007/3-540-58855-8_1.

M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000. doi: 10.1023/A:1010071910869.

M. J. Wooldridge. *Agent technology: foundations, applications, and markets*. Springer Science & Business Media, 1998.

M. J. Wooldridge. *Reasoning about rational agents*. MIT press, 2000.

E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, Department of Computer Science, University of Toronto, Canada, 1995.

T. Yue, L. C. Briand, and Y. Labiche. A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation. In *Model Driven Engineering Languages and Systems*, pages 484–498. Springer, 2009.

T. Yue, L. C. Briand, and Y. Labiche. An automated approach to transform use cases into activity diagrams. In *Modelling Foundations and Applications*, pages 337–353. Springer, 2010.

F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003. doi: 10.1145/958961.958963.

CHAPTER A

User Study on Activity Diagrams

College Human Ethics Advisory Network (CHEAN)
College of Science, Engineering and Health

Email: seh-human-ethics@rmit.edu.au
Phone: [61 3] 9925 4620
Building 91, Level 2, City Campus/Building 215, Level 2, Bundoora West Campus

26 October 2016

Associate Professor John Thangarajah
School of Science
RMIT University

Dear A/Prof Thangarajah

ASEHAPP 108-16 User study on specifying requirements in The Prometheus Methodology

The Science, Engineering and Health College Human Ethics Advisory Network (CHEAN) of RMIT University thank you for providing a full copy of your approved Human Research Ethics application together with a complete copy of the University of Otago Human Research Ethics Committee approval documentation for your research project titled: ***User study on specifying requirements in The Prometheus Methodology***.

With research projects that involve applications to more than one Human Research Ethics Committee (HREC), the Science, Engineering and Health College Human Ethics Advisory Network (CHEAN) adopts the following policy:
When an RMIT researcher is involved in a research project which has been approved by a non-RMIT human ethics committee, the ethics application does not need to be reviewed at RMIT, it does however need to be 'registered' at RMIT.

The approving human ethics committee will remain responsible for the oversight of the project and in the case or your research it is noted that the University of Otago Human Research Ethics Committee is the committee responsible for the oversight of your project and has the primary ethical duty of care over the research participants.

Your University of Otago approved Human Research Ethics application has been **accepted** by the CHEAN and your **registration** number is **ASEHAPP 108-16.** Please refer to this number in any future correspondence to the CHEAN regarding this project.

Pre-Expriment Questionnaire

The main purpose of the pre-questionnaire is to assess the user's experience in the filed of AOSE, specifically the Prometheus methodology. Also, it aims to assess your experience in the UML activity diagrams.

About this questionnaire

On a scale of 1-5 (with 1 being Very Unfamiliar, and 5 being Very Familiar) Please rate your familiarity with:

1a. Software Requirements Analysis Concepts: Vary Unfamiliar **1 2 3 4 5** Very Familiar

1b. Intelligent Agents: Vary Unfamiliar **1 2 3 4 5** Very Familiar

1c. The Prometheus AOSE Methodology: Vary Unfamiliar **1 2 3 4 5** Very Familiar

On a scale of 1-5 (with 1 being inexperienced, and 5 being expert) Please rate your familiarity with:

2a. Software Analysis and Design : inexperienced **1 2 3 4 5** expert

2b. UML Behavioural Models: inexperienced **1 2 3 4 5** expert

2c. Use Case Scenarios: inexperienced **1 2 3 4 5** expert

2c. BDI-Agent System Modelling: inexperienced **1 2 3 4 5** expert

3. How many software systems have you designed (not including agent-oriented)?
- ☐ 0
- ☐ 1-5
- ☐ 6-10
- ☐ 11-20
- ☐ 21-29
- ☐ 30+

4. How many agent software systems have you designed?
- ☐ 0
- ☐ 1-5
- ☐ 6-10
- ☐ 11-20
- ☐ 21-29
- ☐ 30+

5. Do you have any expertise in other AOSE methodologies other than Prometheus? List them if yes
- ☐ No
- ☐ Yes ________________________________

User Study on specifying requirements in the Prometheus methodology

Yoosef

November 7, 2014

Participant information

You are invited to participate in a research project being conducted by a PhD student at RMIT University. Your participation is entirely voluntary. This information sheet describes the project in plain, simple language. Please read this sheet carefully and be confident that you understand its contents before deciding whether to participate. If you have any questions about the project, please ask the investigator.

- **Project Title**
  User Study on specifying requirements in the Prometheus methodology.
- **Investigators:**
  Dr. John Thangarajah, PhD. Primary Supervisor. John.thangarajah@rmit.edu.au, 9925 9535
  Dr. James Harland, PhD. Associate Supervisor. james.harland@rmit.edu.au, 9925 2045
  Dr. Tim Miller, PhD. Associate Supervisor. tmiller@unimelb.edu.au, 8344 1318
  Mr. Yoosef Abushark, PhD student and investigator, yoosef.abushark@rmit.edu.au, 0432076551.
- **Who is involved in this research project? Why is it being conducted?**
  - Yoosef Abushark is a PhD student at RMIT University, and he is working in the area of agent-oriented software engineering. He is supervised by Dr. John Thangarajah, Ass.Pro. James Harland and Dr. Tim Miller.
  - This project is part of a PhD thesis, and it proposes a new approach for specifying requirements in the Prometheus methodology. We are conducting the current study to determine how well the proposal affects the way of specifying requirements in the Prometheus methodology.
  - The project has been approved by the RMIT Human Research Ethics Committee.
- **Why have you been approached?**
  You have been approached either because we know you and think you may be interested, or willing to assist us, or because you have registered to do small tasks for payment and have

chosen, or are considering choosing, this task. We have no specific selection criteria beyond the need to be an adult, and to understand English.

- **What is the project about?**
  This project aims to enhance the way of specifying requirements in the Prometheus methodology. The main research question we are trying to address is:
  - Does the proposed approach **enhance** the agents requirements analysis in the following aspects:
    - ∗ Design Understandability: To measure how both approaches affect the understandability of the designers in relation to the agent detailed models.
    - ∗ Artefacts Consistency: To investigate whether both approaches enable designers to pick the missing entities in the top level of the design (e.g. goal overview and analysis overview), and how they affect them in incorporating these changes (addition, removal and/or modification).
    - ∗ Maintainability: To measure how easy to incorporate the changes in both approaches.
  - We are hoping to have fifteen people assess both approaches, and to help us understand what is working well and what needs further work.
- **If you agree to participate, what will you be required to do?**
  Your role is to help us assess both approaches (i.e. we are not testing you). You cannot do anything wrong. We will be asking you to try achieving number of tasks (by an editable PDF) on a given use case scenario that requires you to walk through the existing approach in the methodology, and with the proposed approach (activity diagram). You do not have to worry about making any mistakes. The session is so that we can enhance the way of specifying requirements in the Prometheus methodology, so we need to hear your honest reactions. The investigator will be with you throughout the session. If you have any questions as you go along, just ask him. Since we are interested in how people use the approaches when they dont have anybody sitting next to them to help, the investigator may not be able to answer your questions right away. But when the session is finished, your questions will be answered. The investigator will be asking you to do some specific tasks. The tasks will be written down for you to read, and will also be read out to you by the investigator. As you try to achieve the tasks, the investigator will ask you to try and think out loud: to say what you are trying to do, and what you are thinking.

  We will also ask you to fill in a questionnaire at the start and end of the session. The session should not take more than 30 minutes to complete. You can take a break during the session at any time.
- **What are the possible risks or disadvantages?**
  - There are no risks or disadvantages in participating, beyond giving up a small amount of your time.
  - Although there are no foreseeable risks, if you are unduly concerned about your responses to any of the questionnaire items or if you find participation in the project distressing, you

should contact John Thangarajah as soon as convenient. He will discuss your concerns with you confidentially and suggest appropriate follow-up, if necessary.

- **What are the benefits associated with participation?**
  There are no immediate benefits to you from your participation in this study, although you might feel more adept with both approaches after exploring it in this session. If you wish, I will send you a summary of findings and recommendations when I complete this project.

- **What will happen to the information you provide?**

  - The information you provide will be anonymous. It will be used only for statistical analysis of what is working well and what needs further attention.
  - The results of this study will be presented in a paper for publication. All reporting of observations and comments will be anonymous, so that you cannot be identified. The research data will be kept securely at RMIT for 5 years after publication, before being destroyed. Potential future uses of the data include improvement of the system and further publications.
  - Because of the nature of data collection, we are not obtaining written informed consent from you. Instead, we assume that you have given consent by your completion and return of the survey.

- **Security of the data**
  Due to the nature of the experiment, we will be asking you to do the tasks on the provided hard copy survey. After completing all experiments, all responses will be, manually, dumbed into a google sheet (google drive). Then, all the hard copies will be destroyed. Regarding the pre and the post questionnaires, we will be using google forms. Thus, all the responses, by default, will be collected in a google sheet on google drive. The drive's account belongs to the RMIT University (@rmit), and hence all the data will be under RMIT University control.

- **What are your rights as a participant?**
  As a participant, you have the right to:

  - Withdraw your participation at any time, without prejudice.
  - Have any unprocessed data withdrawn and destroyed, provided it can be reliably identified as yours.
  - Have any questions answered at any time.

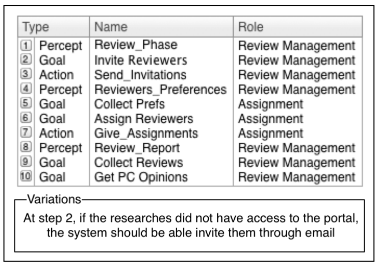
- **Who should you contact if you have any questions?**
  If you require any further information, you may contact the investigator (Yoosef Abushark).

- **What other issues should you be aware of before deciding whether to participate?**
  There are no other issues you should be aware of.

Expriment Prerequisite Materials *(Optional)*

- **Specifying Requirements in the Prometheus methodology**
  In Prometheus, requirements are specified using goals and use case scenarios. A scenario is a sequence of steps that describe a particular run of the system. These steps are of different types ranging from sub-scenarios to the goals that need to be achieved by the system. Although a scenario is a single sequence of steps, the result is a set of sequences, because a goal step can be realised by implementing its children (more generally, its descendants), or its parent from the goal model of the system (goal overview diagram). For example the scenario in Figure 1a, the "Invite_Reviewers" goal step could be realised either by the goal step itself or along with "Invite_Reviewers_Via_Email"; or " Invite_Reviewers_Via_Portal" goal steps. Also, a scenario may have some variations at some point. Such variations are specified in a free-text manner. consequently, these variations may introduce new goals and/or modifying the compositions of existing goals in the goal overview diagram.

  - *Review scenario of the conference management system*
    The conference management system is an agent-based system that helps in managing the different phases of the international conferences including: submission, review, decision and paper collection.In the submission phase, the system should be able to assign a number for each submission and provide receipts to authors. After the specified submission deadline, the system assigns papers to the reviewers. After receiving the reviews, a decision should be made, by the system, about accepting or rejecting the papers with notifying authors. Then, the system collects the accepted papers and prints them in a form of a conference proceeding. As Figure 1 shows, the review scenario (Figure 1a)

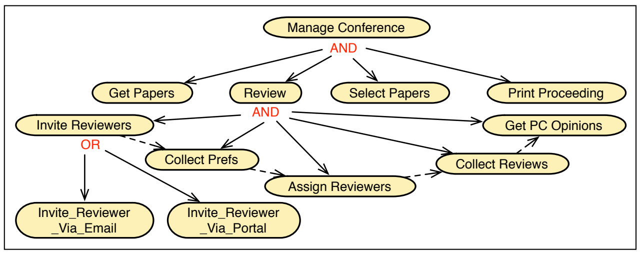


Type		Name	Role
1	Percept	Review_Phase	Review Management
2	Goal	Invite Reviewers	Review Management
3	Action	Send_Invitations	Review Management
4	Percept	Reviewers_Preferences	Review Management
5	Goal	Collect Prefs	Assignment
6	Goal	Assign Reviewers	Assignment
7	Action	Give_Assignments	Assignment
8	Percept	Review_Report	Review Management
9	Goal	Collect Reviews	Review Management
10	Goal	Get PC Opinions	Review Management

Variations

At step 2, if the researches did not have access to the portal, the system should be able invite them through email

(a) Scenario Description



(b) Goal Overview Diagram

Figure 1: Review Scenario & System's Goal Overview Diagram

includes number of steps ranging from percepts the system perceives to goals to pursue. Also, one text-based variation is specified after the second step. Figure 10b outlines the goals and sub-goals required to successfully review the papers. As can be seen in the figure, there are three types of goal decomposition: OR, undirected-AND (denoted by AND) and directed-AND (denoted by AND with dashed arrows between the child goals indicating the ordering constraints, e.g. Invite Reviewers to Collect Prefs).

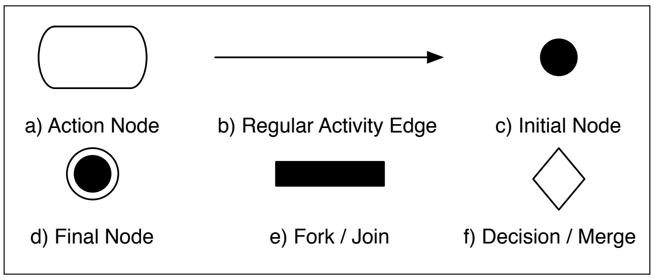


Figure 2: Adopted UML activity digram notations

- **UML Activity Diagrams**
  UML activity diagrams are typically used for business process modelling, for modelling the logic captured by a single use case scenario. Activity diagrams provide designers with a range of graphical notations for modelling activity diagrams including: activity nodes and activity edges. The following six notations (Figure 2) are necessary for this project:

  (a) *Action Node:* it is the fundamental and the executable node in the activity diagrams. It is notated by a rectangular shape with rounded edges.

  (b) *Regular Activity Edge:* It is a directed connection that shows the flow between the different actions within an activity diagram.

  (c) *Initial Node:* It is a control node that initiates the execution of an activity.

  (d) *Final Node:* It is a control node that shows the end of an activity.

  (e) *Fork / Join:* It is control node that splits the execution flow into multiple threads to be executed independently, and then synchronises them.

  (f) *Decision / Merge:* It is control node that splits the execution of an activity into multiple alternate flows with only one flow to be executed.

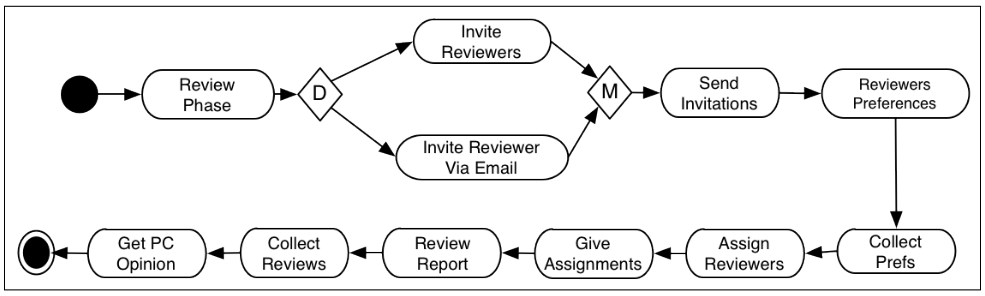


Figure 3: Equivalent AD for the scenario in 1a

Revisiting the conference management system example, Figure 3 shows the equivalent activity diagram for the scenario along with its variation. In fact, Activity Diagram is a coherent structure that merges the scenario steps along with their related information from the goal overview diagram

Trading Agent System

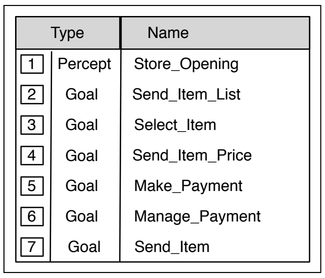
The trading agent system is an agent-based system that captures the process that takes place in a sale transaction. The system has three agents including: seller, buyer and banker. The seller agent must send the list of products to the buyer agent when it receives the ”store_opening” percept. The buyer agent then selects a product. After that, the seller agent should send the buyer the price of the selected item.

Then, the buyer agent should proceed with the payment through the banker agent. The banker agent then processes the payment and notifies both the seller and the buyer about the payment process outcomes (approved or denied). The order of these notifications is not important (e.g. seller first and buyer second or the other way around). In the case of an approved payment, the seller must send the item to the buyer.

NOTE : The provided designs across the tasks are possible designs and do not necessarily represent good ones.
NOTE Activity Diagram is a coherent structure that merges the scenario steps along with their related information from the goal overview diagram.
Remember: A goal step can be realised through its children, or in some cases through its parent

Task 1

Assuming that the scenario description (Figure 4) along with the goal overview (Figure 5) capture the specifications of the system, determine the valid and invalid traces.
Remember: Activity diagram acts as a coherent structure that merges the steps of the scenario along with its related information from the goal overview diagram, and hence it can be used in achieving the task.



	Type	Name
1	Percept	Store_Opening
2	Goal	Send_Item_List
3	Goal	Select_Item
4	Goal	Send_Item_Price
5	Goal	Make_Payment
6	Goal	Manage_Payment
7	Goal	Send_Item

Figure 4: Sale Transaction Scenario Description

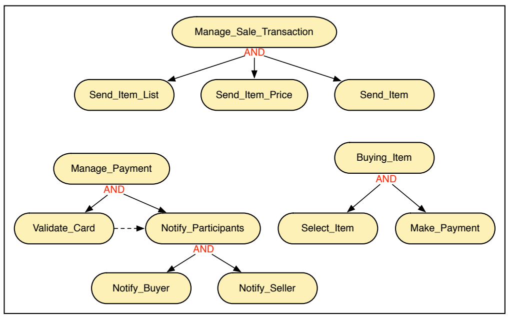


Figure 5: System Goal Overview Diagram

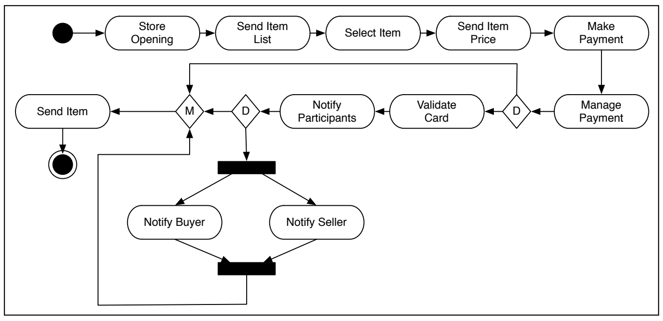
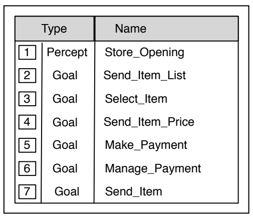


Figure 6: Activity Diagram

	Execution Trace 1	Execution Trace 2	Execution Trace 3	Execution Trace 4	Execution Trace 5
Token 1	Store_Opening	Store_Opening	Store_Opening	Store_Opening	Store_Opening
Token 2	Send_Item_List	Send_Item_List	Send_Item_List	Send_Item_List	Send_Item_List
Token 3	Select_Item	Buying_Item	Select_Item	Select_Item	Select_Item
Token 4	Send_Item_Price	Send_Item_Price	Send_Item_Price	Send_Item_Price	Send_Item_Price
Token 5	Make_Payment	Make_Payment	Make_Payment	Make_Payment	Make_Payment
Token 6	Validate_Card	Manage_Payment	Manage_Payment	Manage_Payment	Manage_Payment
Token 7	Notify_Participants	Validate_Card	Validate_Card	Validate_Card	Validate_Card
Token 8	Send_Item	Notify_Participants	Notify_Participants	Notify_Participants	Notify_Participants
Token 9	-	Send_Item	Send_Item	Notify_Seller	Notify_Buyer
Token 10	-	-	-	Notify_Buyer	Notify_Seller
Token 11	-	-	-	Send_Item	Send_Item
Answer:	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid

Task 2

According to the basic run (scenario in Figure 7), “Manage_Payment”, “Send_Item” is one of the possible paths after posting “Make_Payment” goal. Specify, if possible, another three paths by numbering the tokens based on their order (refer to the example column).
Remember: Activity diagram acts as a coherent structure that merges the steps of the scenario along with its related information from the goal overview diagram, and hence it can be used in achieving the task.



	Type	Name
1	Percept	Store_Opening
2	Goal	Send_Item_List
3	Goal	Select_Item
4	Goal	Send_Item_Price
5	Goal	Make_Payment
6	Goal	Manage_Payment
7	Goal	Send_Item

Figure 7: Sale Transaction Scenario Description

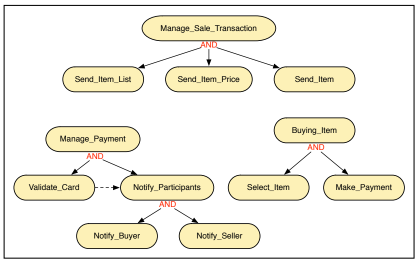


Figure 8: System Goal Overview Diagram

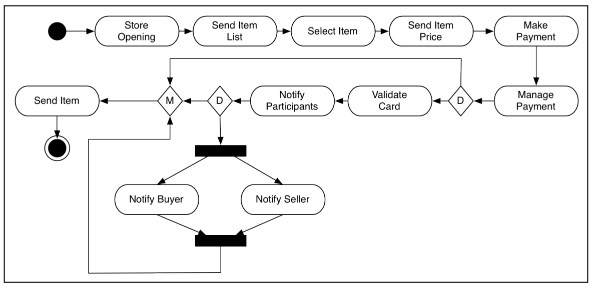


Figure 9: Activity Diagram

Example (this path is based on the basic run)	Path 1 ☐ Not Possible	Path 2 ☐ Not Possible	Path 3 ☐ Not Possible
(___)Manage Sale Transaction	(___)Manage Sale Transaction	(___)Manage Sale Transaction	(___)Manage Sale Transaction
(___)Validate Card	(___)Validate Card	(___)Validate Card	(___)Validate Card
(___)Notify Seller	(___)Notify Seller	(___)Notify Seller	(___)Notify Seller
(___)Notify Participants	(___)Notify Participants	(___)Notify Participants	(___)Notify Participants
(___)Notify Buyer	(___)Notify Buyer	(___)Notify Buyer	(___)Notify Buyer
(___)Buying Item	(___)Buying Item	(___)Buying Item	(___)Buying Item
(___)Send Item List	(___)Send Item List	(___)Send Item List	(___)Send Item List
(___)Select Item	(___)Select Item	(___)Select Item	(___)Select Item
(___)Send Item Price	(___)Send Item Price	(___)Send Item Price	(___)Send Item Price
(___)Make Payment	(___)Make Payment	(___)Make Payment	(___)Make Payment
(**1**)Manage Payment	(___)Manage Payment	(___)Manage Payment	(___)Manage Payment
(**2**)Send Item	(___)Send Item	(___)Send Item	(___)Send Item
(___)Store Opening	(___)Store Opening	(___)Store Opening	(___)Store Opening

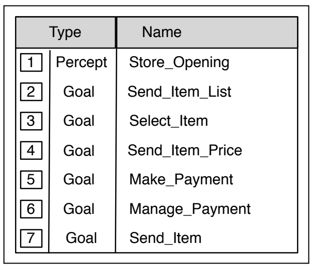
Task 3

One of the stakeholders wants the buyer agent to be able to show its disinterest in a product. Thus, a variation to the scenario is appeared as follows:

After the Buyer agent receives the item_list event from the Seller agent, it may select a product or may **show** its **disinterest**

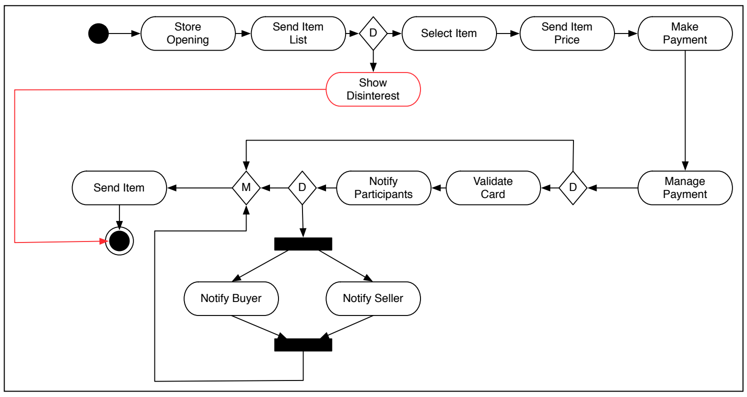
Your task is to add the new goal/s introduced by the variation in the goal overview diagram (Figure 11).

Remember: Activity diagram acts as a coherent structure that merges the steps of the scenario along with its related information from the goal overview diagram, and hence it can be used in achieving the task.



	Type	Name
1	Percept	Store_Opening
2	Goal	Send_Item_List
3	Goal	Select_Item
4	Goal	Send_Item_Price
5	Goal	Make_Payment
6	Goal	Manage_Payment
7	Goal	Send_Item

(a) Scenario



(b) Activity Diagram

Figure 10: Scenario & Activity Diagram

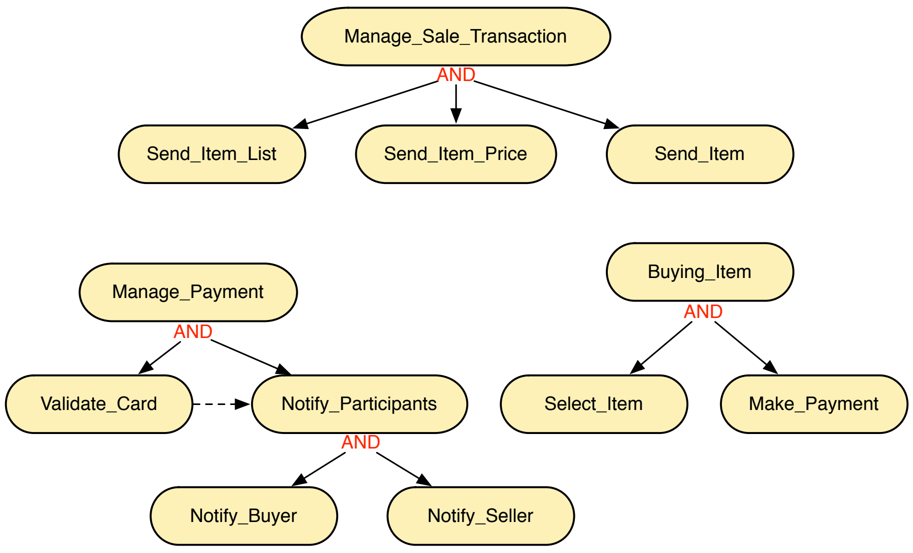


Figure 11: System Goal Overview Diagram

Task 4

Considering the two traces below, the appearance of Trace 2 is due to a change requested by the stakeholders. Your task is to annotate the activity diagram in Figure 14 to capture both traces in the table below.

Token#	Trace 1	Trace 2
1	Store_Opening	Store_Opening
2	Send_Item_List	Send_Item_List
3	Select_Item	Select_Item
4	Send_Item_Price	Send_Item_Price
5	Make_Payment	Make_Payment
6	Manage_Payment	Manage_Payment
7	Send_Item	Cancel_Order

	Type	Name
1	Percept	Store_Opening
2	Goal	Send_Item_List
3	Goal	Select_Item
4	Goal	Send_Item_Price
5	Goal	Make_Payment
6	Goal	Manage_Payment
7	Goal	Send_Item

Figure 12: Scenario Description

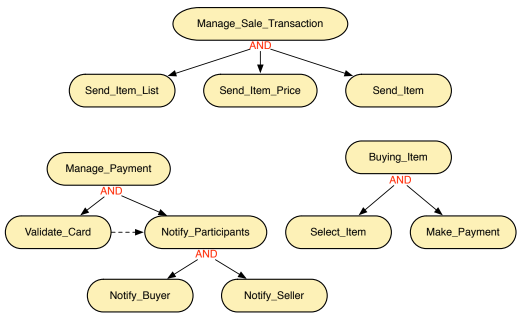


Figure 13: System's Goal Overview

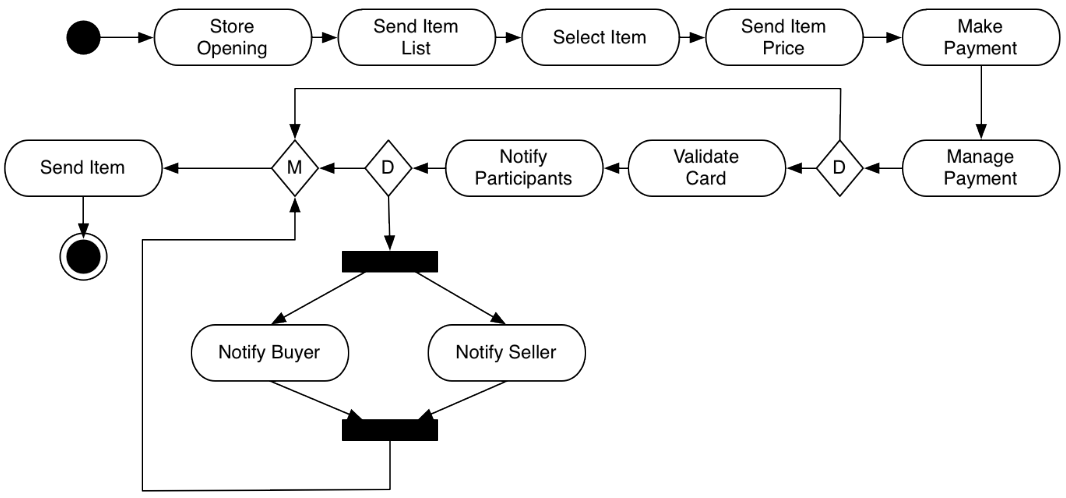


Figure 14: Activity Diagram

Auction System

The auction system is a small agent-based system that simulates the activities that take place in an auction. The system has five agents including: auctioneer, three bidders and banker. The Auction has seven items to bid upon (one item at each round).

The Auctioneer agent should announce the start of the action to the bidders after it receives the "new_auction percept from the environment. The percept includes information about the item to bid and its reserve value.

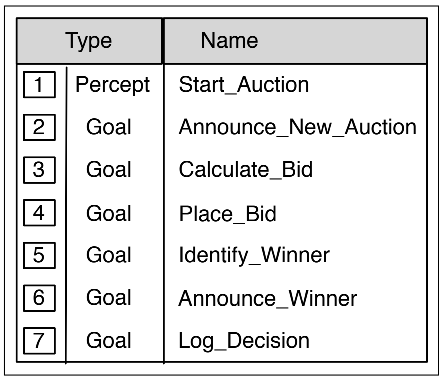
Then, the bidders should calculate and place their bids. After that, the auctioneer agent should decide on the winning bid and notify the bidders about its decision (who won). Bidders should then update their beliefs with such decision (either winning or lost).

NOTE : The provided designs across the tasks are possible designs and not necessarily represent good ones.

Remember: A goal step can be realised through its children, or in some cases through its parent

Task 1

Assuming that the scenario description below along with the goal overview (Figure 16) capture the specifications of the system, determine the valid and invalid traces.



	Type	Name
1	Percept	Start_Auction
2	Goal	Announce_New_Auction
3	Goal	Calculate_Bid
4	Goal	Place_Bid
5	Goal	Identify_Winner
6	Goal	Announce_Winner
7	Goal	Log_Decision

Figure 15: Auction Scenario Description

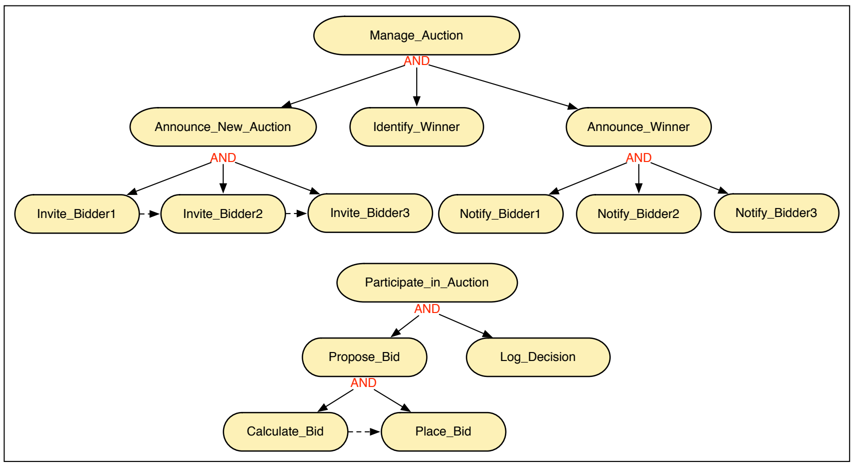
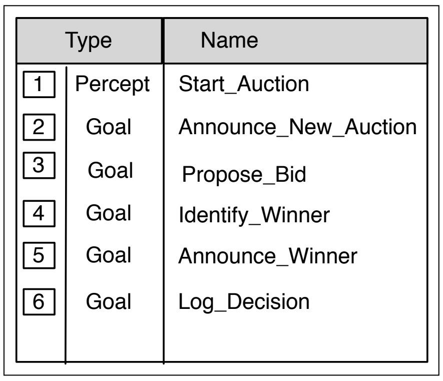


Figure 16: System Goal Overview Diagram

	Execution Trace 1	**Execution Trace 2**	**Execution Trace 3**	**Execution Trace 4**	**Execution Trace 5**
Token 1	Start_Auction	Start_Auction	Start_Auction	Start_Auction	Start_Auction
Token 2	Announce_New_Auction	Announce_New_Auction	Announce_New_Auction	Announce_New_Auction	Announce_New_Auction
Token 3	Calculate_Bid	Invite_Bidder1	Propose_Bid	Calculate_Bid	Invite_Bidder1
Token 4	Place_Bid	Invite_Bidder2	Calculate_Bid	Place_Bid	Invite_Bidder3
Token 5	Identify_winner	Invite_Bidder3	Place_Bid	Identify_Winner	Invite_Bidder2
Token 6	Announce_Winner	Calculate_Bid	Identify_Winner	Announce_Winner	Propose_Bid
Token 7	Log_Decision	Place_Bid	Announce_Winner	Notify_Bidder1	Calculate_Bid
Token 8	-	Identify_Winner	Notify_Bidder1	Notify_Bidder3	Place_Bid
Token 9	-	Announce_Winner	Notify_Bidder3	Notify_Bidder2	Identify_Winner
Token 10	-	Log_Decision	Notify_Bidder2	Log_Decision	Announce_Winne
Token 11	-	-	Log_Decision	-	Log_Decision
Answer:	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid	☐ Valid ☐ Invalid

Task 2

According to the basic run (scenario in Figure 17), “Announce_New_Auction”, “Propose_Bid”, “Identify_Winner”, “Announce_Winner” and “Log_Decision” is one of the possible paths after receiving “Start_Auction” percept. Specify, if possible, another three paths by numbering the tokens based on their order (refer to the example column).



Type		Name
1	Percept	Start_Auction
2	Goal	Announce_New_Auction
3	Goal	Propose_Bid
4	Goal	Identify_Winner
5	Goal	Announce_Winner
6	Goal	Log_Decision

Figure 17: Auction Scenario Description

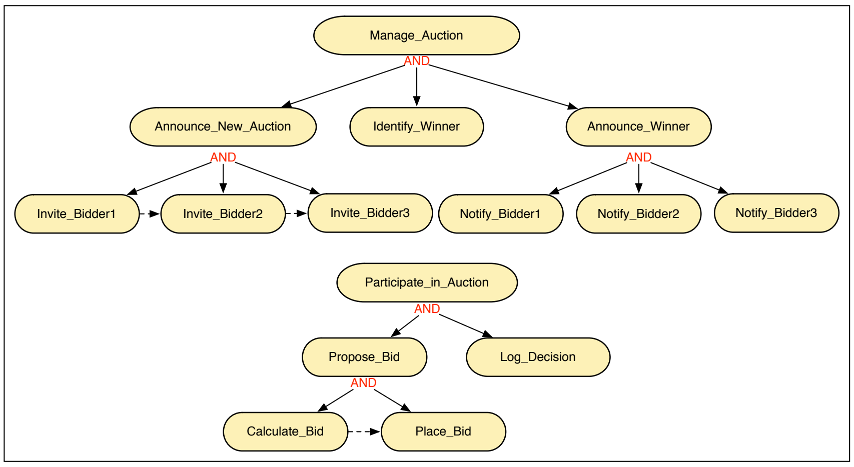


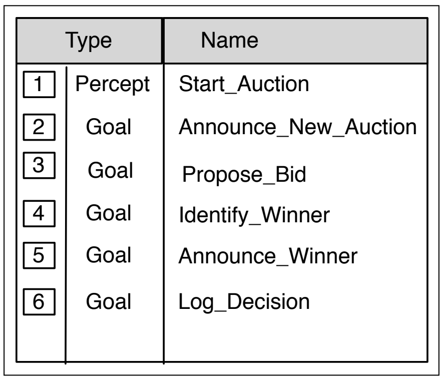
Figure 18: System Goal Overview Diagram

Example (this path is based on the basic run)	Path 1 ☐ Not Possible	Path 2 ☐ Not Possible	Path 3 ☐ Not Possible
(___)Manage_Auction	(___)Manage_Auction	(___)Manage_Auction	(___)Manage_Auction
(**1**)Announce_New_Auction	(___)Announce_New_Auction	(___)Announce_New_Auction	(___)Announce_New_Auction
(___)Invite_Bidder1	(___)Invite_Bidder1	(___)Invite_Bidder1	(___)Invite_Bidder1
(___)Invite_Bidder2	(___)Invite_Bidder2	(___)Invite_Bidder2	(___)Invite_Bidder2
(___)Invite_Bidder3	(___)Invite_Bidder3	(___)Invite_Bidder3	(___)Invite_Bidder3
(**3**)Identify_Winner	(___)Identify_Winner	(___)Identify_Winner	(___)Identify_Winner
(**4**)Announce_Winner	(___)Announce_Winner	(___)Announce_Winner	(___)Announce_Winner
(___)Notify_Bidder1	(___)Notify_Bidder1	(___)Notify_Bidder1	(___)Notify_Bidder1
(___)Notify_Bidder2	(___)Notify_Bidder2	(___)Notify_Bidder2	(___)Notify_Bidder2
(___)Notify_Bidder3	(___)Notify_Bidder3	(___)Notify_Bidder3	(___)Notify_Bidder3
(___)Participate_in_Auction	(___)Participate_in_Auction	(___)Participate_in_Auction	(___)Participate_in_Auction
(**2**) Propose_Bid	(___)Propose_Bid	(___)Propose_Bid	(___)Propose_Bid
(___)Calculate_Bid	(___)Calculate_Bid	(___)Calculate_Bid	(___)Calculate_Bid
(___)Place_Bid	(___)Place_Bid	(___)Place_Bid	(___)Place_Bid
(**5**) Log_Decision	(___) Log_Decision	(___) Log_Decision	(___) Log_Decision
(___)Start_Auction	(___)Start_Auction	(___)Start_Auction	(___)Start_Auction

Task 3

One of the stakeholders wants the bidder agents to be able to request a loan from the banker agent in the case of three successive losses.. Thus, a variation to the scenario in Figure 19 is appeared as follows:

In the case when a bidder agent successively loses the auction three times, it needs to **apply for bank loan**, rather than proposing a new bid.



Type		Name
1	Percept	Start_Auction
2	Goal	Announce_New_Auction
3	Goal	Propose_Bid
4	Goal	Identify_Winner
5	Goal	Announce_Winner
6	Goal	Log_Decision

Figure 19: Auction Scenario Description

Your task is to add the new goal/s introduced by the variation above in the goal overview diagram.

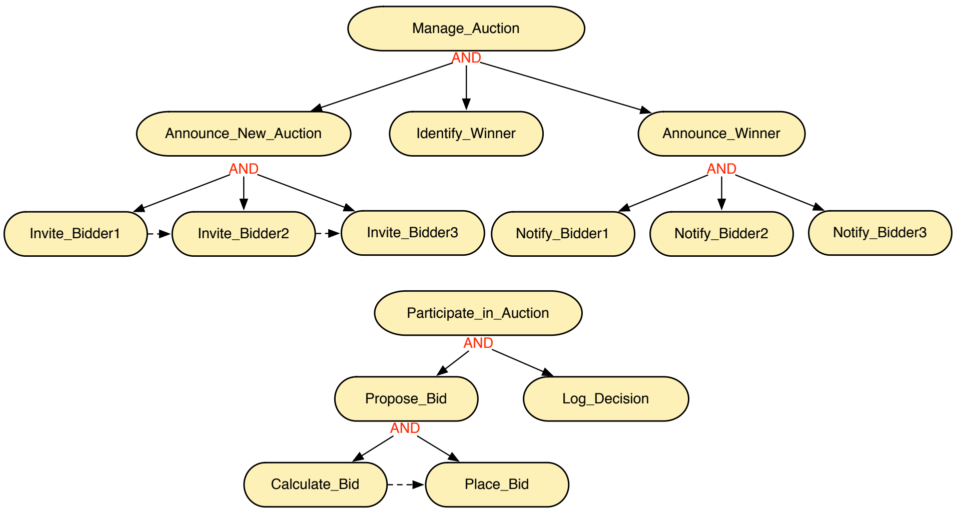


Figure 20: System Goal Overview Diagram

Post-Expriment Questionnaire

This post test questionnaire is designed to get the user's feedback on their experience in both approaches.

Using the following rating sheet, please select the number closest to the term that most closely matches your feeling about the activity diagram as an extra artefact along with scenario and goal overview diagram.

1. Enables an easy extraction of a possible behaviour path relative to a particular scenario

[ ]Strongly Agree [ ]Agree [ ] Neutral [ ]Disagree [ ]Strongly Disagree

2. The time taken to grasp what the entire activity diagram shows is reasonable

[ ]Strongly Agree [ ]Agree [ ] Neutral [ ]Disagree [ ]Strongly Disagree

3. It is easy to maintain activity diagrams (easy to incorporate changes including: adding, removing and modifying entities)

[ ]Strongly Agree [ ]Agree [ ] Neutral [ ]Disagree [ ]Strongly Disagree

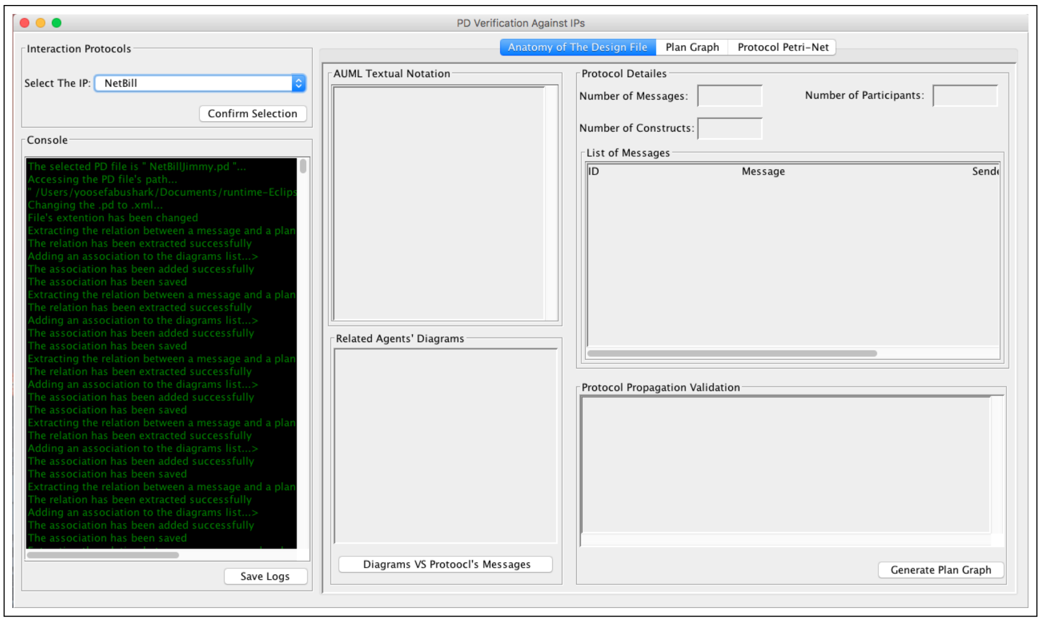
4. Activity Diagram is useful in designing the agents of a particular scenario.

[ ]Strongly Agree [ ]Agree [ ] Neutral [ ]Disagree [ ]Strongly Disagree

In the experiment you followed two approaches in specifying requirements including: the approach without activity diagram and the one with activity diagram. Which Approach would you prefer to use? and why.

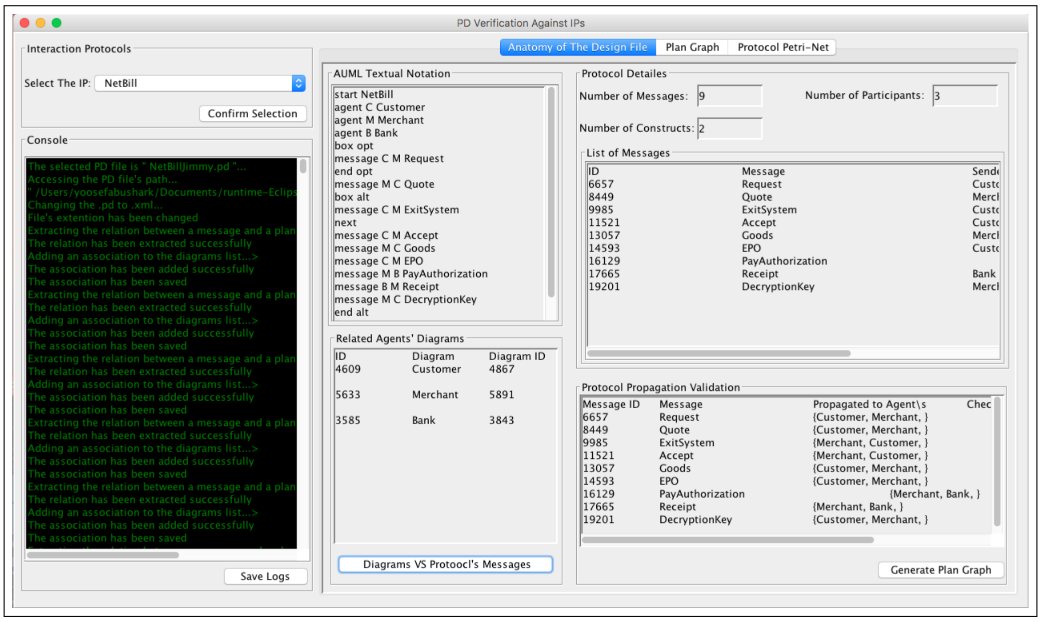
CHAPTER B

Screenshots of The Tool-Support



```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

Figure B.1: "Design v.s. interaction protocols" Tool after accessing the verification feature from the "Package Explorer" within PDT



```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

```
start NetBill
agent C Customer
agent M Merchant
agent B Bank
box opt
message C M Request
end opt
message M C Quote
box alt
message C M ExitSystem
next
message C M Accept
message M C Goods
message C M EPO
message M B PayAuthorization
message B M Receipt
message M C DecryptionKey
end alt
```

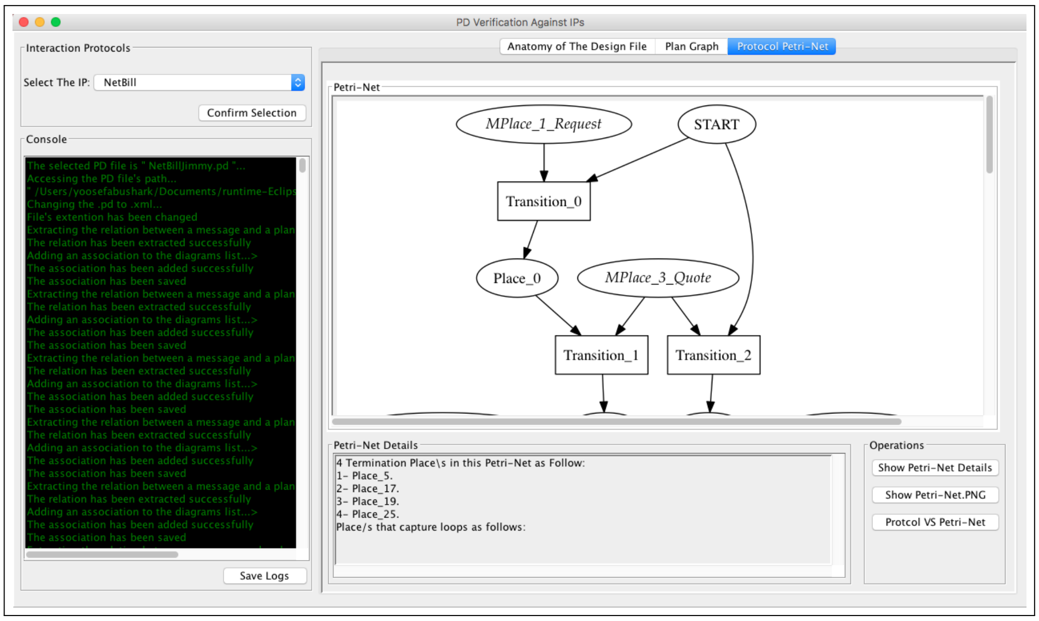
ID	Diagram	Diagram ID
4609	Customer	4867
5633	Merchant	5891
3585	Bank	3843

Number of Messages: 9
Number of Participants: 3
Number of Constructs: 2

ID	Message	Send
6657	Request	Custo
8449	Quote	Merch
9985	ExitSystem	Custo
11521	Accept	Custo
13057	Goods	Merch
14593	EPO	Custo
16129	PayAuthorization	
17665	Receipt	Bank
19201	DecryptionKey	Merch

Message ID	Message	Propagated to Agent\s	Chec
6657	Request	{Customer, Merchant, }	
8449	Quote	{Customer, Merchant, }	
9985	ExitSystem	{Merchant, Customer, }	
11521	Accept	{Merchant, Customer, }	
13057	Goods	{Customer, Merchant, }	
14593	EPO	{Customer, Merchant, }	
16129	PayAuthorization	{Merchant, Bank, }	
17665	Receipt	{Merchant, Bank, }	
19201	DecryptionKey	{Customer, Merchant, }	

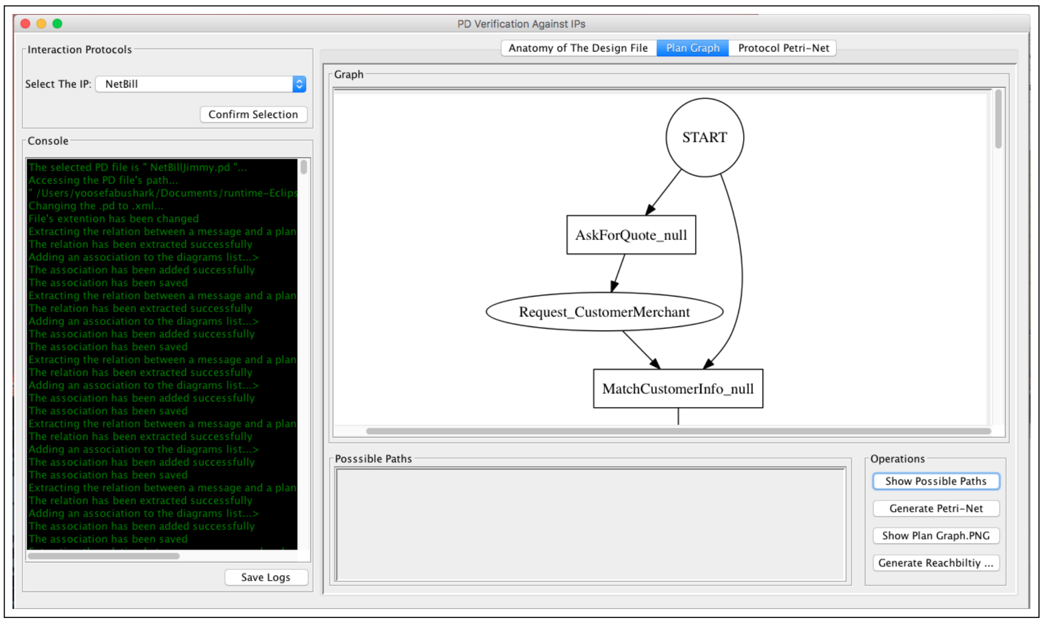
Figure B.2: The tool interface after selecting and confirming a protocol from the drop-down list.



```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

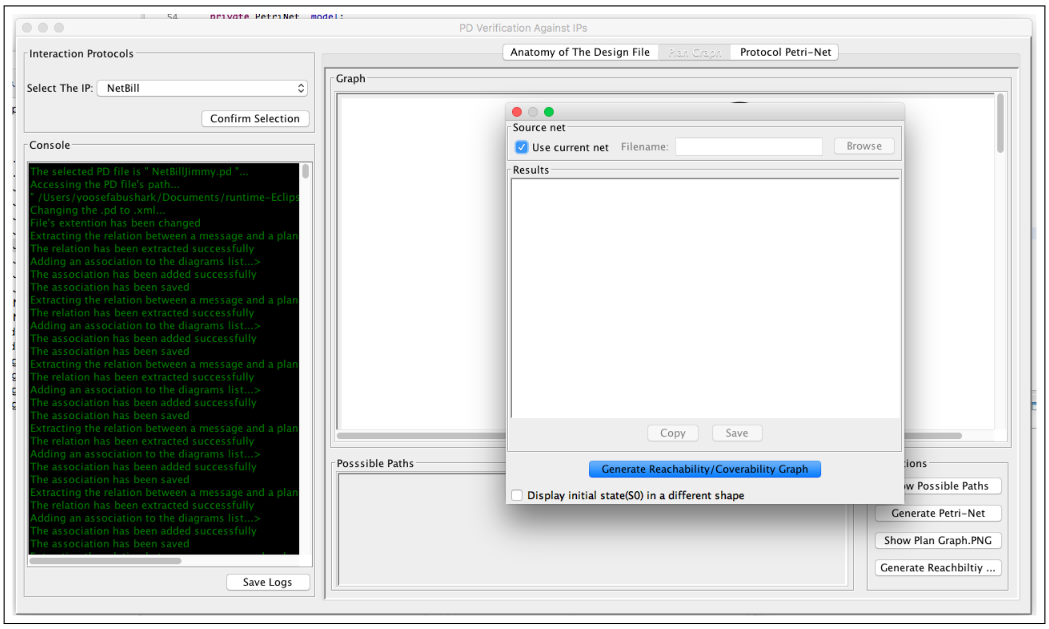
```
4 Termination Place\s in this Petri-Net as Follow:
1- Place_5.
2- Place_17.
3- Place_19.
4- Place_25.
Place/s that capture loops as follows:
```

Figure B.3: The tool interface after generating the Plan Graph.



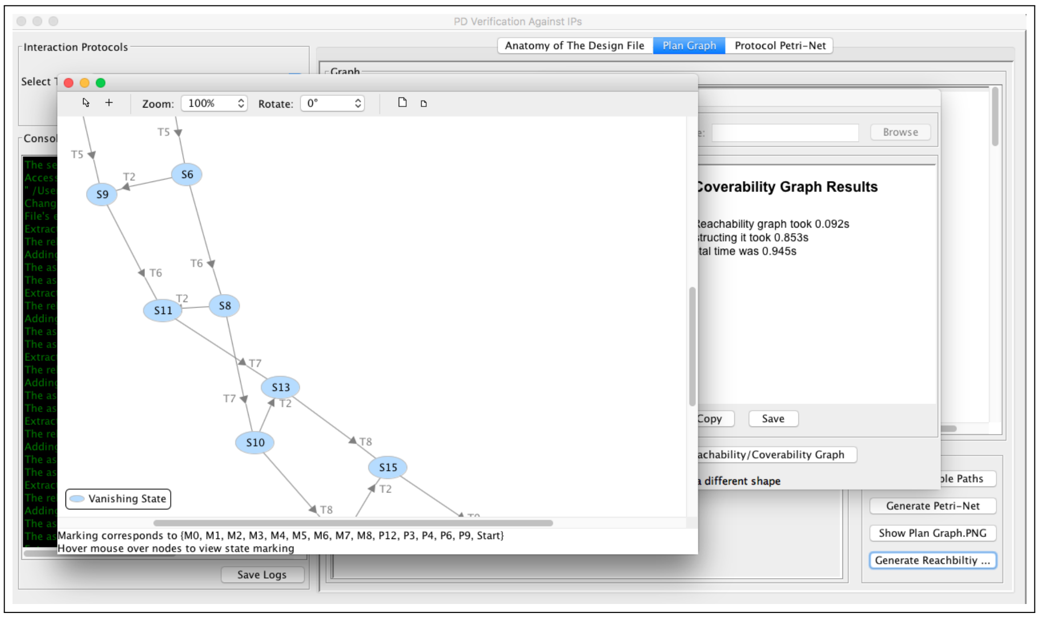
```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

Figure B.4: The tool interface after generating the Petri net of the selected protocol.



```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

Figure B.5: The tool interface after translating the plan graph to Petri net to calculate its reachability graph.

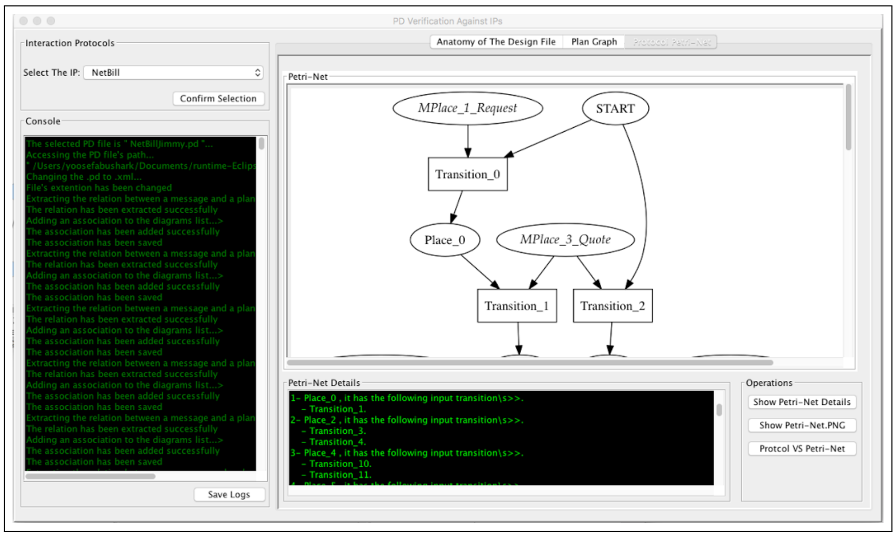


Coverability Graph Results

eachability graph took 0.092s
structing it took 0.853s
tal time was 0.945s

Marking corresponds to {M0, M1, M2, M3, M4, M5, M6, M7, M8, P12, P3, P4, P6, P9, Start}
Hover mouse over nodes to view state marking

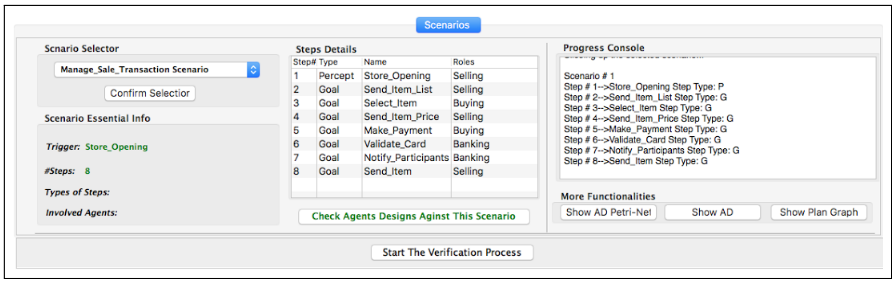
Figure B.6: The tool interface after calculating the reachability graph of the Petri net equivalent to the plan graph.



```
The selected PD file is " NetBillJimmy.pd "...
Accessing the PD file's path...
" /Users/yoosefabushark/Documents/runtime-Eclips
Changing the .pd to .xml...
File's extention has been changed
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
Extracting the relation between a message and a plan
The relation has been extracted successfully
Adding an association to the diagrams list...>
The association has been added successfully
The association has been saved
```

```
1- Place_0 , it has the following input transition\s>>.
   - Transition_1.
2- Place_2 , it has the following input transition\s>>.
   - Transition_3.
   - Transition_4.
3- Place_4 , it has the following input transition\s>>.
   - Transition_10.
   - Transition_11.
```

Figure B.7: The tool interface after finishing off the checking process and generating the textual report.



Trigger: Store_Opening

#Steps: 8

Types of Steps:

Involved Agents:

Step#	Type	Name	Roles
1	Percept	Store_Opening	Selling
2	Goal	Send_Item_List	Selling
3	Goal	Select_Item	Buying
4	Goal	Send_Item_Price	Selling
5	Goal	Make_Payment	Buying
6	Goal	Validate_Card	Banking
7	Goal	Notify_Participants	Banking
8	Goal	Send_Item	Selling

```
Scenario # 1
Step # 1-->Store_Opening Step Type: P
Step # 2-->Send_Item_List Step Type: G
Step # 3-->Select_Item Step Type: G
Step # 4-->Send_Item_Price Step Type: G
Step # 5-->Make_Payment Step Type: G
Step # 6-->Validate_Card Step Type: G
Step # 7-->Notify_Participants Step Type: G
Step # 8-->Send_Item Step Type: G
```

Figure B.8: “Design v.s. Requirements” Tool after accessing the verification feature from the “Package Explorer” within PDT.

CHAPTER C

Aircraft Turned-Around Simulator Case-Study

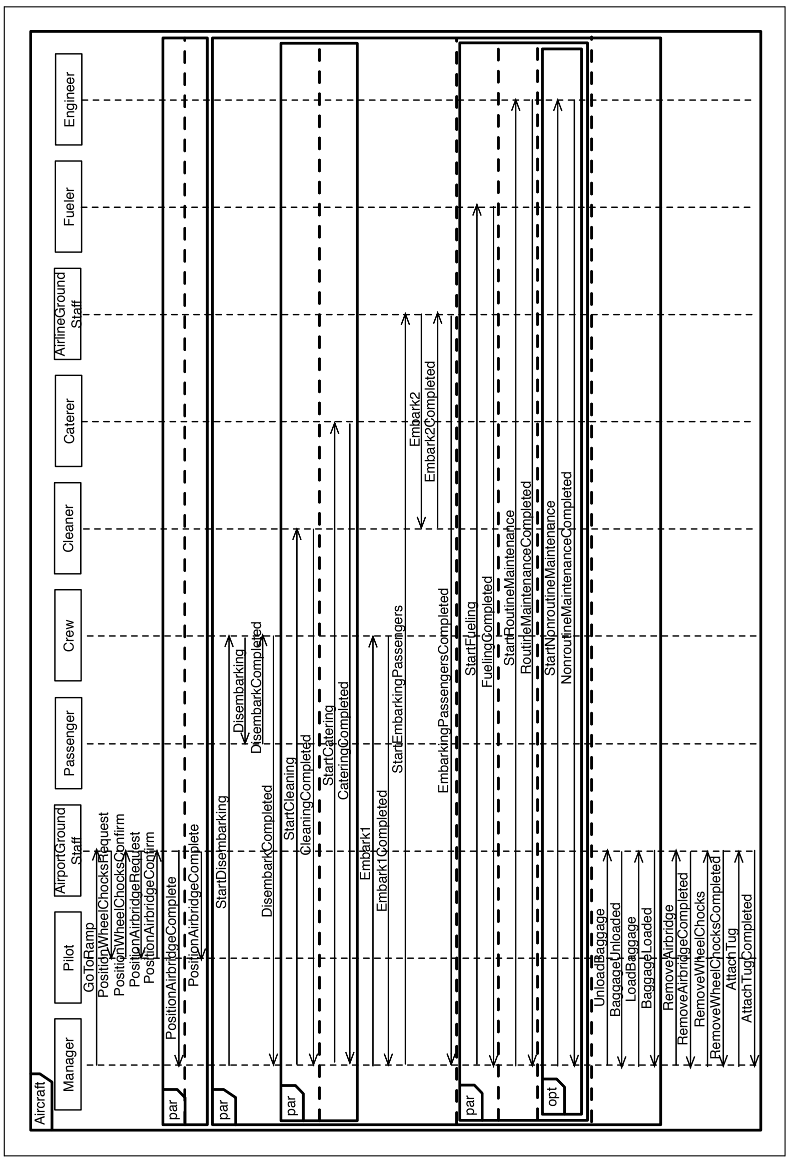
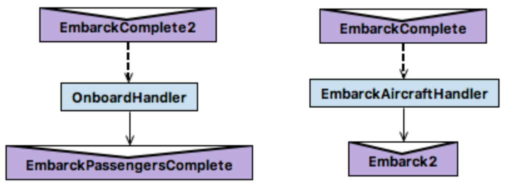


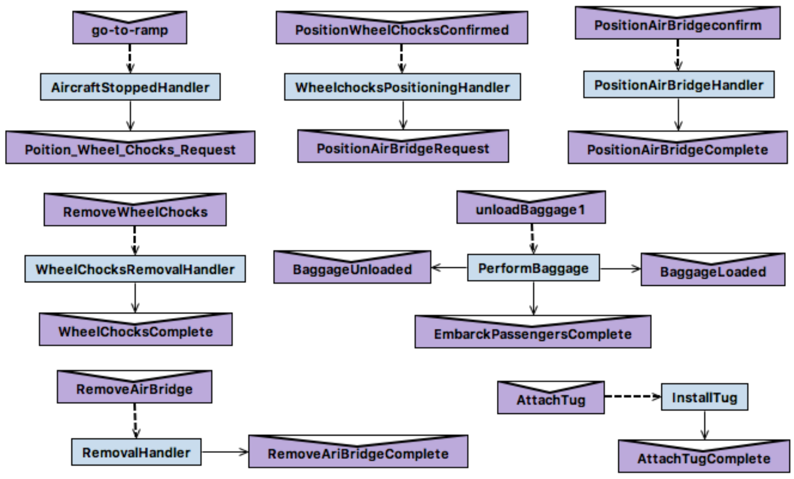
Figure C.1: Simulation Protocol

ATS

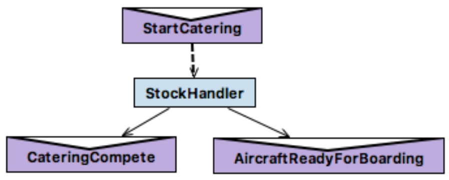
1 Airline Ground Staff Agent



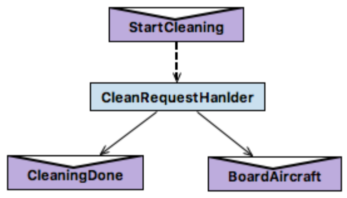
2 Airport Ground Staff Agent



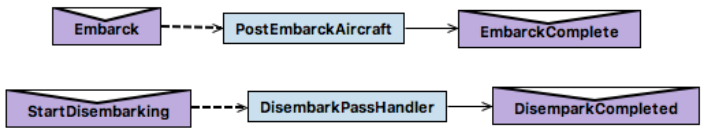
3 Caterer



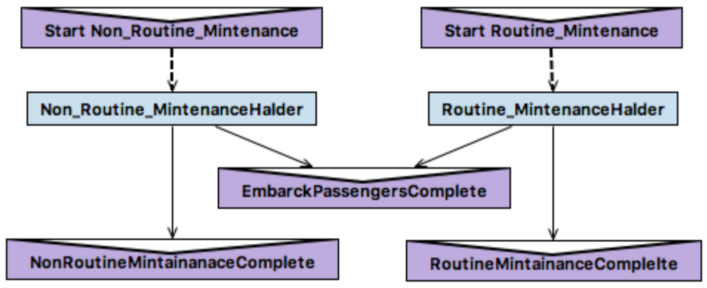
4 Cleaner



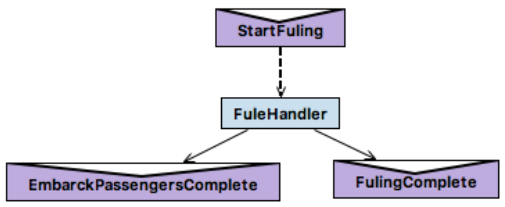
5 Crew



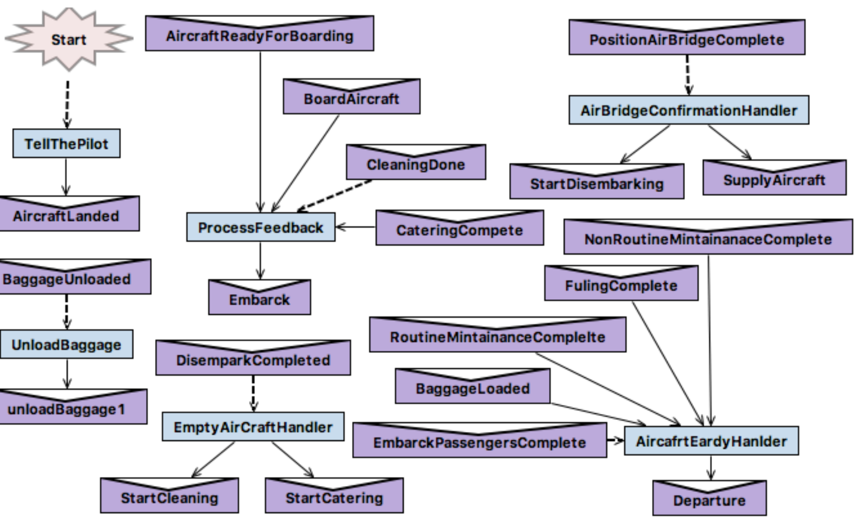
6 Engineer


Start Non_Routine_Mintenance
Non_Routine_MintenanceHalder
Start Routine_Mintenance
Routine_MintenanceHalder
EmbarckPassengersComplete
NonRoutineMintainanaceComplete
RoutineMintainanceComplelte

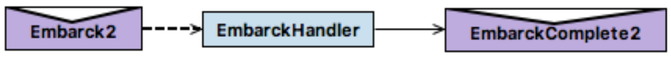
7 Fuller



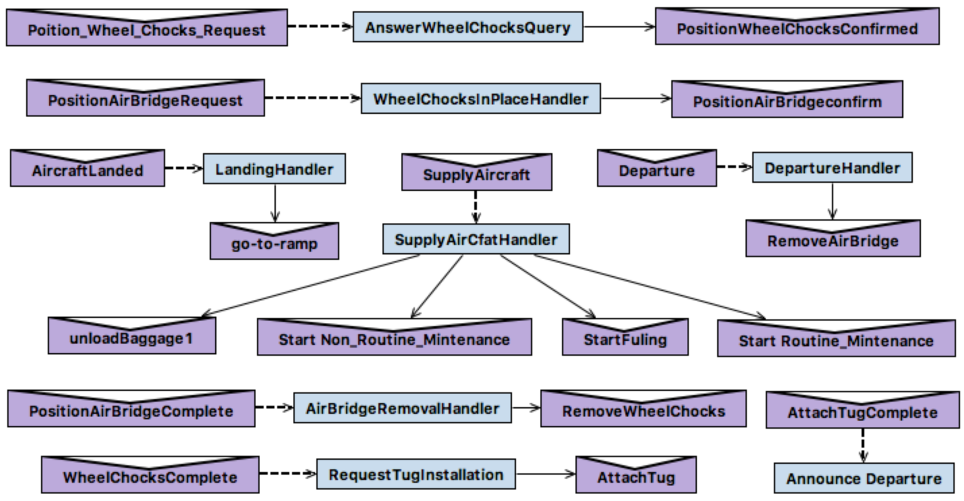
8 Manager



9 Passengers



10 Pilot



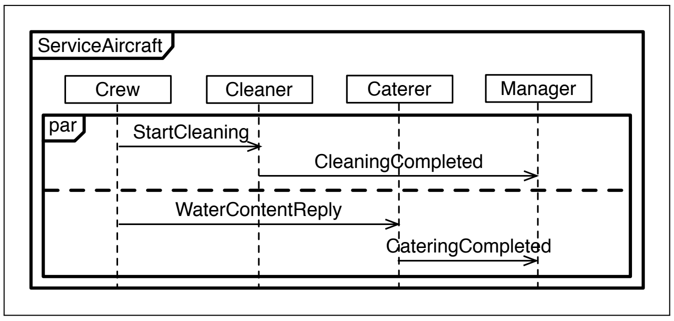


Figure C.2: ServiceAircraft Protocol

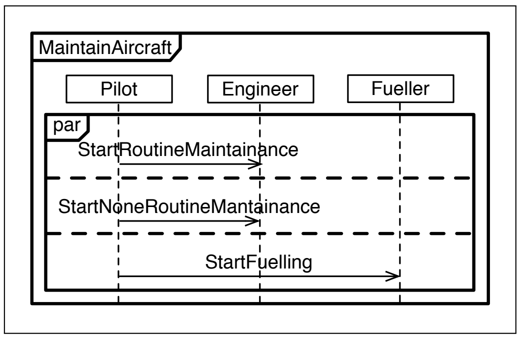


Figure C.3: MaintainAircraft Protocol

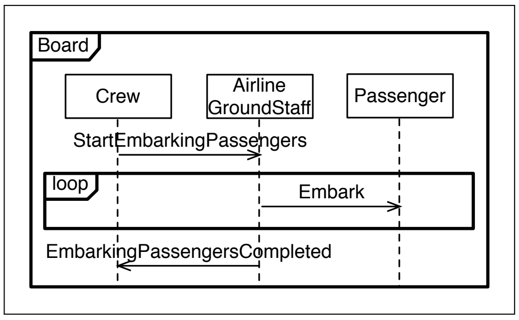


Figure C.4: Board Protocol

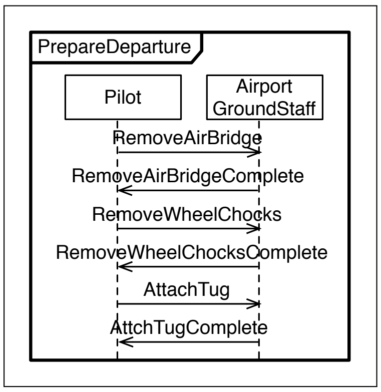


Figure C.5: PrepareDeparture Protocol

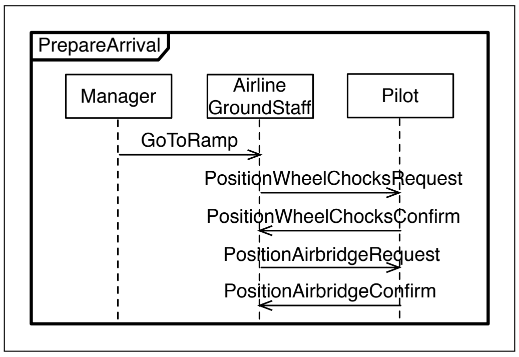


Figure C.6: PrepareArrival Protocol

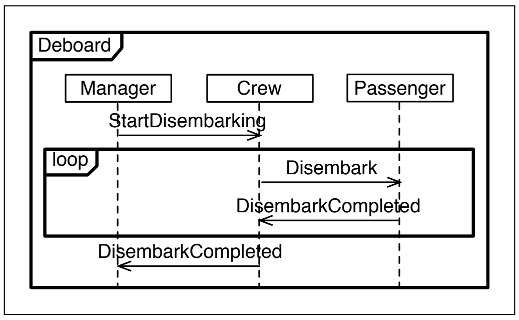


Figure C.7: Deboard Protocol

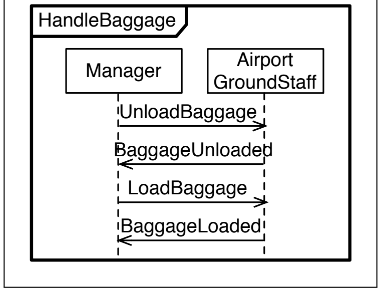


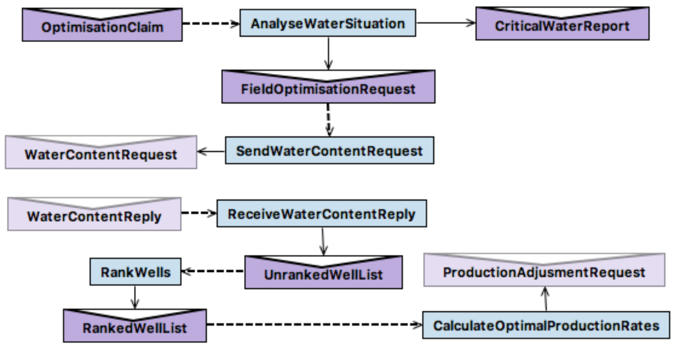
Figure C.8: HandleBaggage Protocol

CHAPTER D

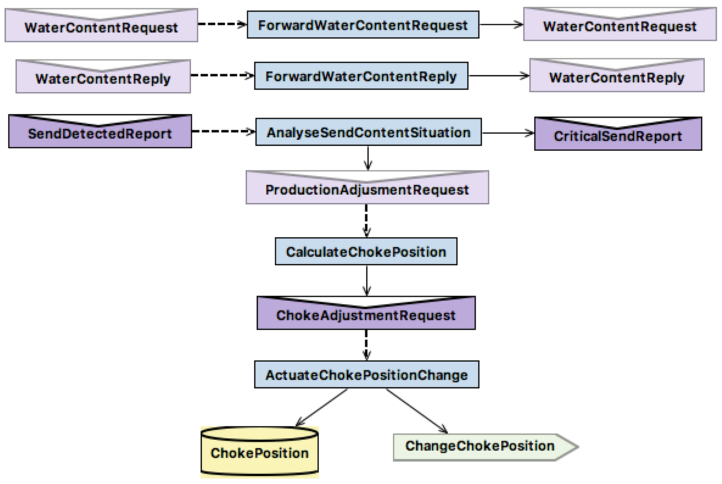
Oil Production Simulation Case-Study

Oil Production

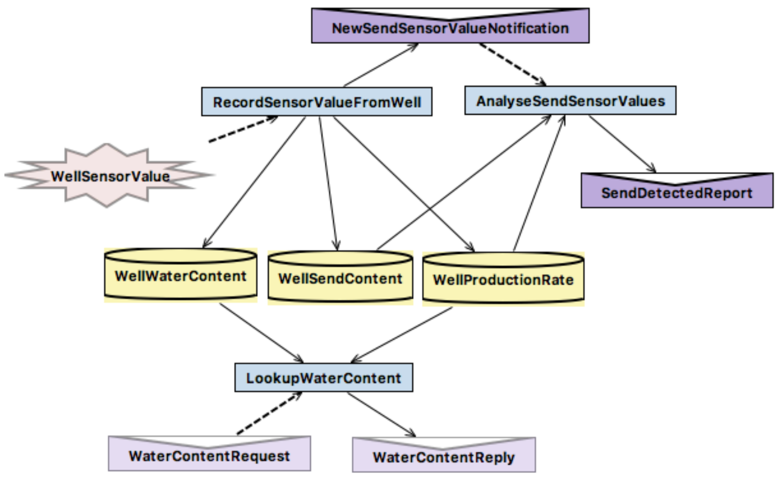
1 Production Optimiser Agent



2 Well Controller Agent



3 Well Monitor Agent



Type		Name	Role
1	Percept	Plant Sensor Values	PlantMonitoring
2	Goal	Record Well Sensor Values	PlantMonitoring
3	Goal	Detect Water Content Change	WellMonitoring
4	Goal	Detect Critical Situation	CriticalSituationDetection
5	Goal	Produce Alarm	AlarmCreation
6	Action	Display Alarm	AlarmPresentation
7	Goal	Notify Operator	AlarmPresentation

Figure D.1: ControlSendContent Scenario

Type		Name	Role
1	Percept	Operator's Choke Adjustment	InstructionAcceptance
2	Goal	Accept Instruction	InstructionAcceptance
3	Action	Change Choke Position	ChokeAdjusment

Figure D.2: RespondToOperator Scenario

Type		Name	Role
1	Percept	Well Sensor Values	WellMonitoring
2	Goal	Record Well Sensor Values	WellMonitoring
3	Goal	Calculate Optimal Choke Position	FelidOptimisation
4	Goal	Detect Critical Situation	CriticalSituationDetection
5	Goal	Produce Alarm	AlarmCreation
6	Action	Display Alarm	AlarmPresentation
7	Goal	Notify Operator	AlarmPresentation

Figure D.3: OptimiseOilProduction Scenario