Practical methodologies for agent-oriented conceptual modelling

A. Krishna

University of Wollongong, aneesh@uow.edu.au
UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
Practical Methodologies for Agent-Oriented Conceptual Modelling

A thesis submitted in fulfillment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Aneesh Krishna

School of Information Technology and Computer Science
June 2006
© Copyright 2006

by

Aneesh Krishna

All Rights Reserved
Dedicated to
Rev. Mamajee & Mameejee
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

______________________________________________
Aneesh Krishna
June 20, 2006
Abstract

Many modelling techniques tend to address “late-phase” requirements while many critical modelling decisions (such as determining the main goals of the system, how the stakeholders depend on each other, and what alternatives exist) are taken during early-phase requirements engineering. The i* modelling framework is a semiformal agent-oriented conceptual modelling language that is well-suited for answering these questions. This dissertation addresses two key challenges faced in the practical deployment of agent-oriented conceptual modelling frameworks such as i*. First, very little exists by way of principled methodologies for eliciting agent-oriented conceptual models. The high-level and abstract nature of these models poses special challenges in devising elicitation techniques. Second, there has been limited adoption of agent-oriented conceptual modelling techniques in industry.

We address the first challenge by developing a novel elicitation technique based on requirements capture templates. These templates can themselves be derived from enterprise/domain ontologies and organisational models, and interact with these in interesting ways.

Our approach to addressing the second problem is based on the observation that the value of conceptual modelling in the i* framework lies in its use as a notation complementary to existing requirements modelling and specification languages, i.e., the expressive power of i* complements rather than supplants that of existing notations. The use of i* in this fashion requires that we define methodologies that support the co-evolution of i* models with more traditional specifications. This research uses the notion of co-evolution in a very specific sense to describe a class of methodologies that permit i* modelling to proceed independently of specification in a distinct notation, while maintaining some modicum of loose coupling via consistency constraints. This research examines how this might be done with formal specification notations (specifically Z) as well as an industry standard modelling language (UML). Our aim, then, is to support the modelling of organisational contexts, intentions and rationale in i*,
while traditional specifications of functionality and design proceeds in either the formal notation or UML. Much of this research has been motivated by issues arising from an industry-scale enterprise modelling exercise conducted for an emergency services agency.

This research has been validated through a detailed case study involving a major government emergency services agency (who provided funding for this project). A very large-scale and comprehensive agent-oriented conceptual model of this agency and its requirements has been developed, and forms the basis for the validation component of this dissertation.
I would like to thank my supervisor, Professor Aditya K. Ghose, for his excellent guidance and advice throughout this thesis. His expertise in artificial intelligence and software engineering research has been most beneficial to my work. I am particularly thankful for his interest in my work on agent-oriented conceptual modelling, the time he was prepared to spend with me in the many meetings we had, and his invaluable feedback on the various drafts of this document.

I would also like to thank my co-supervisor Associate Professor Peter Hyland for his helpful comments and advice. Further thanks go to all the members of the Decision Systems Lab at the School of Information Technology and Computer Science, University of Wollongong for creating an interesting and stimulating research environment. This project was mainly funded by a Australian Postgraduate Award (Industry) scholarship from Australian Research Council grant involving NSW State Emergency Service (SES), Australia. I thank Mrs. Kim McCall, Manager-Information Technology at SES for her help and support during the project.

Finally, I would like to thank my parents, Mr. Shiam Krishna Saxena, Mrs. Asha Saxena, my sisters Mrs. Neelima, Mrs. Anupama, Dr. Pratima, brother-in-laws Dr. Yash, Mr. Desh Ratn, Dr. Susmit, kids and my wife Komal for all their understanding and support.
List of Publications

This is a list of referred papers that is related to this research work.


## Contents

Abstract v
Acknowledgments vii
List of Publications viii

1 Introduction 1

2 Background 5

2.1 Requirements Engineering (RE) .............................. 5

2.2 Goal-oriented RE ................................................. 9

2.2.1 KAOS ....................................................... 10

2.2.2 The NFR framework .......................................... 12

2.2.3 Other Approaches ............................................. 13

2.3 Agent-Oriented Conceptual Modelling ......................... 15

2.3.1 The $i^*$ framework ........................................... 17

2.3.2 Agent-Object Relationship (AOR) modelling ............... 21

2.4 Agent-Oriented Software Engineering ........................ 21

2.4.1 Prometheus .................................................... 22

2.4.2 Gaia Methodology .............................................. 23

2.4.3 Other Approaches ............................................. 23

2.5 Early- and Late-phase RE ....................................... 24

2.5.1 Tropos Methodology .......................................... 25

2.5.2 Formal Tropos ................................................ 26

2.5.3 Combined use of $i^*$ and ConGolog ........................ 30

2.5.4 Combined use of $i^*$ and Albert-II ......................... 31

2.5.5 Combined use of $i^*$ with 3APL and AgentSpeak(L) .... 31

2.6 Formal Methods .................................................. 32
## List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Scope of the discussed Approach/Methodology</td>
<td>32</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of Organisational Unit Template for Department Details</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Function Elaboration Template for Function Elaboration</td>
<td>54</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of Activity Elaboration Template for Internal Intentional Characteristics of individual actor(s) to achieve the activity</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Key steps in the proposed methodology of mapping $i^*$ into Z</td>
<td>68</td>
</tr>
<tr>
<td>4.2</td>
<td>Terminology used in arriving at Z schemas</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>Key steps in the proposed methodology of co-evolution of formal and informal requirements</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Key steps in the proposed methodology of mapping $i^*$ model into UML sequence diagram</td>
<td>122</td>
</tr>
<tr>
<td>6.2</td>
<td>Effect annotation of dependencies</td>
<td>126</td>
</tr>
<tr>
<td>6.3</td>
<td>Effect annotations of tasks (internal tasks)</td>
<td>127</td>
</tr>
<tr>
<td>A.1</td>
<td>Organisational Unit Template for Department Details</td>
<td>143</td>
</tr>
<tr>
<td>A.2</td>
<td>Function Elaboration Template for Function Elaboration</td>
<td>144</td>
</tr>
<tr>
<td>A.3</td>
<td>Activity Elaboration Template for Internal Intentional Characteristics</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>of individual actor(s) to achieve the activity</td>
<td></td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A Portion of the KAOS Conceptual Meta-model [32]</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>A partial softgoal hierarchy for usability [98]</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>The Strategic Dependency Model of the Meeting Scheduling System</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>The Strategic Rationale Model of the Meeting Scheduling System</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Fundamental actors of the case study and their goals [55]</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>SD diagram of the Insurance Company case study [55]</td>
<td>28</td>
</tr>
<tr>
<td>2.7</td>
<td>An example of messages being sent between objects [7]</td>
<td>43</td>
</tr>
<tr>
<td>2.8</td>
<td>The system object calling its determineAvailableReports method [7]</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>A Strategic Dependency model for computer based training system</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Strategic Rationale model for computer based training system</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Notion of an Organisational Model</td>
<td>51</td>
</tr>
<tr>
<td>3.4</td>
<td>Ontology-model-template co-evolution</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>The Strategic Dependency Model of the Flood Rescue Management case study</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>The Strategic Rationale Model of the Flood Rescue Management case study</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>The Strategic Rationale Model of the Actor Call Taking Supervisor/System</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>The Strategic Rationale Model of the Actor Volunteers/Emergency Worker</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td>The Strategic Rationale Model of the Actor Emergency Coordination Centre Coordinator</td>
<td>81</td>
</tr>
<tr>
<td>4.6</td>
<td>The Strategic Rationale Model of the Actor Field Control Centre Coordinator</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>The Strategic Rationale Model of the Actor Community</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>Motivations for Co-evolution of Formal and Informal Requirements</td>
<td>98</td>
</tr>
<tr>
<td>5.2</td>
<td>Co-evolution of $i^*$ models and Z specifications</td>
<td>101</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.3 Modified Strategic Dependency Model of the Flood Rescue Management</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>case study</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4 Modified Strategic Rationale Model of the Flood Rescue Management</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>case study</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1 SD model of the computer based training system</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>6.2 SR model of the computer based training system</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>6.3 Computer based training system sequence diagram</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>6.4 Annotated computer based training system sequence diagram</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>7.1 Combined use of proposed methods</td>
<td>135</td>
<td></td>
</tr>
</tbody>
</table>
Requirements Engineering is the process which identifies the purpose of a software system and documents it in a form which is responsive to detailed examination, communication followed by implementation [100]. The criticality and centrality of requirements engineering is now widely acknowledged, given that the majority of software errors can be traced back to errors in the requirements engineering phase, and given the difficulty of recovering from these errors in subsequent phases (i.e., design specification, coding, testing, etc.) of the software life cycle [11]. Conceptual modelling plays an important role in requirements engineering, both by offering the means to represent models in a manner that can be understood by a large and diverse set of stakeholders, and also by offering the possibility of performing certain kinds of analysis on these models very early in the software life cycle.

Agent-Oriented Conceptual Modelling (AOCM) is a novel approach to conceptual modelling that has gained considerable credence within the research community over the last decade [144, 154]. The key innovation in AOCM is the use of the notion of an agent, together with associated concepts such as goals, plans, commitments etc., as modelling constructs. This has been inspired, in part, by the growing popularity of agent-oriented approaches to the building of intelligent systems, within the artificial intelligence, and related research communities. This dissertation focuses on a specific AOCM approach - the $i^*$ modelling framework [154]. AOCM frameworks are of interest for several different reasons. First, they offer modelling constructs at a higher level of abstraction than other existing conceptual modelling notations. In many ways, this parallels the transition from object-oriented programming to agent-oriented programming, with the concomitant benefits that accrue from being able to program at a higher-level of abstraction [157]. Second, they permit greater emphasis to be placed on the early phase of requirements engineering [153]. Much of the existing work in requirements engineering has focused on modelling and specification languages for the late-phase, and assume that an initial statement of the requirements is always available. The
late-phase of requirements engineering focuses on the completeness, consistency, and automated verification of requirements. In contrast, the early phase intends to model and analyse stakeholders' interests and how they might be addressed, or compromised, by means of various system and environment alternatives [156]. AOCM notations offer abstractions that are sufficiently high-level to be particularly suitable to early-phase requirements modelling. Third, they provide an affective means of modelling complex organisational contexts. Fourth, they allow us to build rich models of stakeholder goals and intentions. While goal-oriented requirements engineering has been developed separately, AOCM frameworks such as i* permit us to situate goal representations within richer models of organisational context [160]. Models of stakeholder intentions permit us to answer “why” questions by pointing to these rich models of stakeholder goals and organisational contexts [159]. While existing research on requirements and design rationale also seek to answer these questions, AOCM notations offer a far richer language for modelling rationale.

This dissertation addresses two key challenges faced in the practical deployment of agent-oriented conceptual modelling frameworks such as i*. First, very little exists by way of principled methodologies for eliciting agent-oriented conceptual models. The high-level and abstract nature of these models poses special challenges in devising elicitation techniques. Second, there has been limited adoption of agent-oriented conceptual modelling techniques in industry. Few attempts have been made to deploy agent-oriented conceptual modelling notations (including i*) in industry-scale settings. There also appear to be other impediments to the widespread uptake of such notations within industry. This research has been conducted in the context of an Australian Research Council Linkage project jointly funded with a government-sector emergency services agency to build a comprehensive enterprise model for this agency. This might also represent the earliest attempt at industry-scale deployment of the i* conceptual modelling notation. The emergency services agency offers some unique features, such as an infrastructure that remains dormant for long periods of time but gets activated during an emergency. In these circumstances, representing intentions and organisational context are important for managing change (requirements evolution, design modifications, code updates etc) and supporting business process reengineering [84]. These features make traditional conceptual modelling somewhat difficult to use, but renders the domain eminently suitable for the deployment of agent-oriented conceptual modelling techniques. The notions of requirements/design rationale are well known, yet they are not reflected in the traditional conceptual modelling notations.
This dissertation addresses the first challenge by presenting a novel approach to requirements elicitation in an agent-oriented conceptual modelling context based on the specific outcomes of our experience with early-phase requirements engineering in the emergency services domain. We show how it is possible to use the underlying modelling notation to drive the design of a set of Requirements Capture Templates that can simplify the process of requirements elicitation via stakeholder interaction. In effect, these templates are forms that an analyst/modeller seeks to complete in the course of a stakeholder interview which provide structure and direction to an otherwise unstructured exercise. Once completed, these templates can be (manually) transformed in a relatively straightforward fashion to eventual SD and SR models (i* model). In addition, these completed templates can serve as a structured repository and record of stakeholder interaction that can be revisited whenever requirements need to be renegotiated or revised (for instance, to resolve an inconsistency) or business processes need to be re-engineered. We show that these templates can themselves be derived from enterprise/domain ontologies and organisational models, and interact with these in interesting ways.

We address the second challenge by exploring how the i* framework might complement rather than supplant existing, industry-standard requirements modelling and specification techniques. In so doing, we deviate somewhat from the approach taken by projects such as Tropos [23, 24], which uses the i* notation to represent early- and late-phase requirements, architectures and detailed designs. We believe that the value of conceptual modelling in the i* framework lies in its use as a notation complementary to existing specification languages, i.e., the expressive power of i* complements that of existing notations. The use of i* in this fashion requires that we define methodologies that support the co-evolution of i* models with more traditional specification and modelling languages. We use the notion of co-evolution in a very specific sense to describe a class of methodologies that permit i* modelling to proceed independently of specification in a distinct notation, while maintaining some modicum of loose coupling between them. We attempt to validate this approach by exploring co-evolution of i* models with models in two very different modelling/specification languages. At the formal end of the spectrum, we explore the co-evolution of i* models with formal specifications in Z [132]. Our aim, is to support the modelling of organisational contexts, intentions and rationale in i*, while traditional specifications of functionality and design proceeds in the formal notation. We note that many of the lessons learnt from this exercise apply
to other formal methods as well. At the other end of the spectrum, we explore the co-evolution of $i^*$ models with models in the industry-standard (and largely semi-formal or informal) UML notation. We specifically focus on UML sequence diagrams, where the representational capabilities are clearly complementary to those of $i^*$ ($i^*$ models are sequence-agnostic).

The rest of the thesis is organised as follows. Chapter 2 presents a brief background of the major areas related to our work. These include Requirements Engineering, Goal-oriented RE, Agent-Oriented Conceptual Modelling, Agent-Oriented Software Engineering, Early and Late-phase of RE, Formal Methods and UML (including UML Sequence Diagrams). In Chapter 3 we introduce our proposal of novel requirements elicitation technique for the $i^*$ notation. In the chapter 4, we shall give a proposal about the combined use of agent-oriented conceptual modelling with formal methods. This chapter explores how these two otherwise disparate approaches might be used in a synergistic fashion for requirements engineering. Chapter 5 presents a methodology for supporting the co-evolution of $i^*$ models and Z specifications. This chapter also explores how consistency is preserved during the co-evolution of $i^*$ models and Z specifications. In the Chapter 6, we shall give a proposal about the Combined use of Agent-oriented Conceptual Modelling with the UML Sequence Diagram. Finally, chapter 7 draws conclusions, some lessons learned and identifies future work.
In this chapter, we begin by explaining Requirements Engineering (RE), and research trends in that area. We also survey existing work in several areas related to this research: Goal-oriented RE, Agent-Oriented Conceptual Modelling, Agent-Oriented Software Engineering, Early- and Late-phase of RE techniques, Formal Methods and UML (including UML Sequence Diagrams).

2.1 Requirements Engineering (RE)

According to Sommerville [131], “Software engineering is an engineering discipline which is concerned with all aspects of software production from early stages of system specification through to maintaining the system after it has gone into use”. SE is the study of how to produce complex software systems on time and on budget. It has no physical limitations, which can lead to extremely complex software. As we get better at producing software more and more complex systems are required.

The software development cycle typically involves the following stages:

- **Requirements Specification:** This involves analysing the current problem and proposing a complete specification of the desired external behavior of the software system to be developed.

- **Design:** In preliminary design, one decomposes the software system into its actual constituent components, and then repeatedly decomposes those components into smaller and smaller sub-components until the sub-components are small enough to be solved by a person easily. Detailed design also defines and documents algorithms for each component that will be realised as code.

- **Coding:** The software design is realised as a set of programs or program units. The result of this phase is an executable program.
• **Testing:** First, in unit testing, each coded module of a subcomponent is tested for the presence of bugs and it is ensured that each module behaves according to its specification as defined during detailed design. Then, in integration testing, one interconnects sets of previously tested modules to ensure that the sets behave as well as they did when independently tested, and to ensure that each component integrated from those sub-components behaves according to its specification defined during preliminary design. Finally, in system testing the complete software system, embedded in its actual hardware environment is checked to ensure that it behaves according to the requirements specification.

• **Delivery, production, and deployment:** after the testing stage, the software and the hardware it runs on should be delivered and become operational for the client.

• **Maintenance and enhancement/improvement:** These processes are actually a full development life cycle. For a coding change, coding and the three subsequent testing stages will have to be performed. For a design change, design, coding and the subsequent testing will have to be performed. For a requirement change all the stages will have to be performed.

The success of the software system is mainly measured on the basis of whether it meets the purpose for which it was planned. The demand of high quality software systems has increased significantly. It is widely accepted that the success of the requirements engineering phase is crucial for the success of the whole software development life cycle. All the development activities are highly dependent on the output of this stage. Requirements Engineering (RE) is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation. There are a number of difficulties encountered in the RE process. Stakeholders (including paying customers, users and developers) may be numerous and distributed. Their goals may differ and conflict, depending on their perceptions of the environment in which they work and the tasks they wish to realise. The goals of the stakeholders may not be clear or may be difficult to articulate, and satisfaction of these goals may be constrained by a variety of factors outside their control [100]. In his classic paper on the essence and accidents of software engineering, Brooks stated that “the hardest single part of building a software system is deciding precisely what to build.... Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements” [17]. Improving the overall
quality of requirements is therefore critical. But at the same time it is a difficult goal to achieve.

The definition of Requirement Engineering (RE) has evolved considerably since the mid-1970s when RE was established as a distinct field of investigation and practice. Earlier RE was defined as the initial stage of Software Engineering concerned with the definition of software requirements to be used for restricting and measuring the ultimate software implementation. But, as the research connected with RE matured, the definition of RE has broadened. In [165], RE was defined as “the branch of Software Engineering concerned with real world goals for, services provided by, and constraints of software intensive systems”. While this definition relates mainly to the aspects of software systems, it extends the earlier definition of RE to include aspects of the real world as well.

Recent definitions further extended the scope of RE to describe the process of not just individual software applications but whole organisational systems (e.g. [56, 86, 110, 155]). Research in this area is based on the logic that during the course of designing software systems, requirements engineers intend to improve problematic organisational situations. Hence, as a discipline, RE brings concerns of software engineering nearer to the problems experienced in organisational settings. RE addresses the problems connected with business goals, plans, processes, etc. and systems to be developed or to be evolved in order to achieve organisational objectives [86].

Boehm [11] indicates that requirements analysis and specification is an extremely important stage to ensure the correctness, cost and time effectiveness of the system development exercise. A ‘requirement’ is defined as “something required; something wanted or needed” [146]. The IEEE standard 729 defines it as “(1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or processed by a system to satisfy a contract, standard, specification or other formally imposed document ” [66].

According to Davis [36], the requirements engineering process is a set of activities including “eliciting or learning about a problem that needs a solution, and specifying the external behaviour of a system that can solve the problem”. The aim of RE is to develop a requirements specification, which is a precisely expressed set of agreed descriptions of the requirements by the stakeholders.

Recent studies in this area have emphasized the need for an engineering approach, where models and languages, methods and tools are used to help in the RE activities.
Empirical studies of software development projects have also confirmed the crucial importance of domain knowledge and requirements analysis. The definitions discussed suggest why the process of engineering requirements is so complex. The scope is somewhat broad as it ranges from human organisations to technical artifacts that must be integrated in such organisations, from high-level objectives to operational prescriptions, and from informal to formal. The target system is not just a piece of software, but also comprises the environment that will surround it; the latter is made of humans, devices, and/or other software. The whole system has to be considered under many facets, e.g., socio-economic, physical, technical, operational, evolutionary, and so forth.

Although there is no common definition of the RE process, it is commonly agreed that activities described below represent the key steps [137]. Although these activities are described independently and in a particular order, in practice, they are actually intertwined, and may span the entire software development life cycle.

- **Domain analysis**: During this activity, the existing system in which the software should be built is studied. All the concerned stakeholders are identified and interviewed. Problems and deficiencies in the existing system are identified, opportunities are investigated and broad objectives on the target system are also identified.

- **Elicitation**: In this activity, alternative models for the target system are explored to meet the identified objectives and requirements and statements on components of such models are identified, probably with the help of hypothetical interaction scenarios. Possible alternative models normally define different boundaries between the software-to-be and its environment.

- **Negotiation and agreement**: All possible alternative requirements/statements are evaluated, risks are analysed and “best” possible tradeoffs that receive agreement from all the concerned parties are selected.

- **Specification**: Here, the complete set of requirements and statements are formulated in a precise manner.

- **Specification analysis**: In this activity, specifications are evaluated for deficiencies (such as inadequacy, incompleteness or inconsistency) and for feasibility (in terms of resources required, development costs etc).

- **Documentation**: Various decisions made during the whole process are documented together along with their underlying rationale.
• Evolution: This activity is concerned with situations where, requirements are modified to accommodate corrections, environmental changes, or the addition of new objectives.

2.2 Goal-oriented RE

Based on the complexity of the requirements engineering process, rigorous techniques are required to offer effective support. In the late seventies, several techniques were devised for modelling functional requirements for software systems. They were based on function-oriented notions, supporting concepts such as data and operations. Among the widely used techniques were Entry-Relationship Diagram [25] for data modelling, and Structured Analysis [39] for describing operations. SADT, introduced by Ross and Schoman [123], was probably the first language for modelling requirements. It was based on a data/operation duality principle: data was defined by producing/consuming operations and operations were defined by their input/output data.

The main disadvantage associated with these languages is that the resulting specifications were vague and imprecise. These languages came with semi-formal syntax and a completely informal semantics [45]. They focused on answering the “what-how” questions during the RE but could not capture answer to “why” questions. They also provided no support for analysts to determine whether the requirements specified were adequate for achieving the higher-level objectives that arise naturally in any requirements engineering process [63, 94, 8, 124]. Yue was the first researcher to argue that the integration of explicit goal representations in requirements models provides a criterion for requirements completeness - a set of requirements is complete if it is sufficient to establish the goals it seeks to achieve [152]. Generally speaking, a goal corresponds to an objective the system should achieve through the cooperation of agents in the software-to-be and in the environment [137].

Research in the area of goal-based reasoning has led to two complementary frameworks for integrating goals and goal refinements in requirements models[137]:

• KAOS, which is a formal framework for reasoning about (mostly) functional requirements

• NFR, which is a framework that supports qualitative reasoning about non-functional requirements

We describe this in greater detail below.
2.2.1 KAOS

Goal refinements in the formal framework like KAOS (Knowledge Acquisition in autO-
mated Specification) [31] are captured through AND/OR graph structures borrowed
from problem reduction techniques in artificial intelligence [99]. AND-refinement links
are used to relate a goal to a set of subgoals through the process of refinement. This
means that satisfying all the subgoals in the refinement is a sufficient condition for sat-
sifying the main goal. OR-refinement links are used to relate a goal to an alternative
set of refinements. This means that satisfying one of the refinements is a sufficient
condition for satisfying the main goal. In this framework, a conflict link between goals
is introduced when the satisfaction of one of them may prevent the satisfaction of the
others. Operationalisation links are also introduced to relate goals to requirements on
operations and objects.

This formal framework led to the KAOS methodology for eliciting, specifying, and
analyzing goals, requirements, scenarios, and responsibility assignments [32]. Require-
ments elaboration in KAOS consists of progressively refining high level goals until con-
straints, objects and operations that are assignable to individual agents are obtained.
An agent is an object which is a processor of some actions. Agents thus control state
transitions. As opposed to the other kinds of objects (e.g., entities, relationships and
events), agents can choose their behaviour [50]. Examples of agents are human beings,
physical devices, or programs that exist or are to be developed in the automated part of
the composite system. A goal in KAOS is defined as a non-operational objective that a
composite system should meet. An objective is said to be non-operational if it cannot
be formulated in terms of states that are controlled by some agent. The concepts and
relationships that constitute the KAOS meta-model are illustrated in Figure 2.1.

According to KAOS methodology, an object is a thing of interest in the domain.
A general domain object can be declared in a more specialised way as an entity, relation-
ship or event. Actions are the responsibility of agents, and can modify the state of objects (including agents). A scenario expresses a typical combination of actions expected to take place in the composite system. Goals can be AND/OR reduced into subgoals. Goals often conflict with other goals. Goals concern objects and must be
AND/OR operationalised into constraints (i.e., operational objectives that can be for-
mulated in terms of states controllable by some agent). Alternative arrangements of
actions, agents and objects can be evaluated with regard to the achievement of goals.
In the KAOS language goals are specified using real-time temporal logic. This formal
goal formulation allows reasoning about goal elaborations and formally debugging goal
Figure 2.1: A Portion of the KAOS Conceptual Meta-model [32]
reductions as well as handling inconsistent goals. The KAOS methodology stresses the need to clearly specify and structure goals, whilst it gives considerably less attention to the issue of the initial identification and formulation of goals [32, 35]. The KAOS framework also addresses issues in requirements acquisition, i.e. goal directed [33] [139], scenario-directed [140], viewpoint-directed strategies [141], and the reuse of requirements specifications [92].

2.2.2 The NFR framework

The usefulness of a system is determined partly by its functionality - i.e., what the system does - and partly by global requirements on its development or operational costs, accuracy, performance, security reliability, maintainability, portability, robustness, etc. According to Sommerville [131], “Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, standards etc”. These non-functional (or quality) requirements play a crucial role during the system development. The NFR (Non Functional Requirements) framework introduced the notion of “softgoals” [95, 96]. This is based on the idea that some goals (typically those based on non-functional requirements or quality factors) can rarely be said to be satisfied in a precise sense but function more as optimisation objectives. Instead of goal satisfaction, a notion of goals satisficing needs to be used in this context. The NFR framework uses a repertoire of labels to describe the degree to which a softgoal is satisficed (or otherwise). If a goal is AND-decomposed into subgoals and all subgoals are satisficed, then the goal is satisficeable, but if a subgoal is denied then the goal is deniable. If a goal contributes negatively to another goal and the former is satisficed, then the latter is deniable.

The NFR framework [96, 26, 28] is a comprehensive framework (with associated tool support) for the representation of non-functional requirements in terms of interrelated goals. Such goals can be refined through refinement methods and can be evaluated in order to determine the degree to which a set of non-functional requirements is supported by a particular design via a procedure that propagates labels through the goal decomposition hierarchy. The NFR model consists of goals that represent non-functional requirements (NFR goals), design decisions (satisficing goals), arguments in support or against other goals (argumentation goals), and goal relationships for relating goals to other goals. Design proceeds by refining one or more times each goal, the parent, into a set of offspring goals. In addition to the classical AND/OR decomposition the NFR
framework supports several types of decomposition links between goals. The decomposition links between the parent goal and its offspring describes how the satisficing of the offspring (or failure thereof) relates to the satisficing of the parent goal. Moreover, NFR goals can be refined into satisficing goals making a commitment to a particular implementation. Finally, NFR goals can be refined into argumentation goals, thereby providing evidence for or against the satisficing of a goal. A Contribution link is used to describe the impact one element has on another. This contribution can be negative or positive. The extent of the contribution can be partial (Help or Hurt) or sufficient (Make or Break) based on the concept of satisficing [129]. Correlation links are used to describe the side effects of the existence of one element to others.

The NFR framework components are formally represented in the Telos knowledge representation language [97]. In addition, goal analysis is supported by domain dependent methods for refining goals. Based on the NFR framework, Figure 2.2 represents the softgoal hierarchy for the requirement that the system be highly usable system which might be as important an objective as any of the functional goals. In Figure 2.2 the softgoal “Usability” represents this requirement. The analysis process associated with this softgoal consists of iteratively decomposing the relevant AND/OR relationships or other more loosely defined dependency relations. These relationships are marked with with a + sign, which indicates that one softgoal supports or “positively influences” another. Correlations can be discovered during the construction of softgoal hierarchies by making use of the generic rules that state conditions under which softgoals of the selected type may positively or negatively influence softgoals of another type. The softgoal “User Flexibility” is evidently improved not only by the system attribute “Modularity”, which allows for module substitutions, but also by the system’s ability to allow setting changes. These factors, however, are not necessarily sufficient to satisfy the “User Flexibility” softgoal. This is the reason why +/- labels are used to describe the relationships instead of the AND/OR labels. Figure 2.2 offers only a partial decomposition of the softgoal. Usually, the softgoals Error Avoidance, Information Sharing, and Ease of Learning have their own rich set of alternatives, which can be elaborated through additional refinements [98].

2.2.3 Other Approaches

Coupled with the goal-driven nature of software systems is the way RE itself is performed. RE requires the involvement of various stakeholders (the sponsor organisation, the system developers and users, external regulators etc). There is also much evidence
Figure 2.2: A partial softgoal hierarchy for usability [98]
of the use of strategic and goal driven processes in many kinds of activities that humans perform [34, 120]. This has also given rise to a goal-driven RE philosophy. This goal-driven view of the RE process is also suggested in design problem solving (the research field that deals with the creation and transformation of systems)[130].

The modelling of goals has been proposed during requirements elicitation in order to describe the current organisational behaviour (Goals, Operators, Methods, and Selection rules-GOMS [22], ORDIT [10], $i^*$ [155]) and methodologies for planning, organisation and control of enterprise (Management by Objectives-MBO [93], IE [91], Information Systems Work and Analysis of Changes-ISAC [87], F3 [19]). Goal analysis techniques have been used in the context of requirements negotiation in order to assist reasoning about the need for organisational change and to provide the context within which deliberation occurs during RE (SIBYL [85], REMAP [113]). Modelling of goals has also been used in requirements specification to describe how organisational change can be implemented in terms of the new system’s components by relating business goals to functional and non-functional system specifications in the GBRAM [4] framework. In the context of requirements validation, goal analysis techniques have been employed to define the stakeholder’s criteria against which the system specifications are validated (GQM [5], [147]).

Recently, an interesting method for the scenario based assessment of non functional requirements has been reported in the literature [60]. An approach for validating functional system requirements with the help of scenarios [135] has also been developed. An approach describing the influence of human factors in the requirements engineering process of socio-technical systems design is presented in [59]. Tool to provide traceability between the scenario models and requirements and to facilitate the generation of new scenarios (and scenario variations) has been proposed in [128]. Another tool has been proposed that is used to aid in the validation of non functional system requirements of complex socio technical systems such as the command and control rooms of military vessels [58].

## 2.3 Agent-Oriented Conceptual Modelling

The concept of an agent is becoming more significant in the areas of Artificial Intelligence (AI) and Software Engineering (SE). A variety of systems are now being developed based on the notion of a software agent. Based on the definition by Jennings et al [73]:
• Software agents are situated - they can sense the environment and perform actions that change the environment in some way

• Autonomous - they have control over their own actions and internal states, and can act without direct intervention from humans (or other agents)

• Flexible - responsive to changes in the environment

• Pro-active - they should be able to exhibit opportunistic, goal-directed behaviour and take the initiative where appropriate

• social - they interact with other artificial agents and humans to complete their tasks and help others

A good description on why agent-oriented approaches are becoming popular can be found in [157]. However, we are presenting here comparison between agent-oriented and object-oriented approaches. Agent-orientation is becoming an increasingly popular approach to the conceptualisation and implementation of complex software systems. Agent-Oriented approaches are also becoming popular in modelling formalisms for requirements engineering and design. Intentional concepts such as goals, beliefs, abilities, commitments, etc., provide a higher-level characterisation of the agent (and hence, system) behaviour compared to object-oriented approaches. Software designed with agent-oriented characteristics such as autonomy, social ability, reactivity, proactivity, and communicative and cooperative abilities can sometimes provide greater functionality and higher quality when compared with object-oriented systems [78]. Progress in software engineering over the past two decades has primarily been made possible through the development of increasingly powerful and natural abstractions with which to model and develop complex systems.

The agent metaphor is powerful in modelling organisational contexts. Agent-Oriented Conceptual Modelling (AOCM) in notations such as the i* framework [154] have gained considerable currency in the recent past. Such notations model organisational context and offer high-level social/anthropomorphic abstractions (such as goals, tasks, softgoals and dependencies) as modelling constructs. It has been argued that such notations help answer questions such as what goals exist, how key actors depend on each other and what alternatives must be considered. These notions are missing in traditional object-oriented approaches. In fact, "for complex systems we find that objects, classes and modules provide an essential yet insufficient means of abstraction" [12]. Complex systems require richer problem solving abstractions. Individual objects
represent too fine a granularity of behaviour [73]. Method invocation is too primitive a mechanism for describing the types of interactions that take place in complex systems. The primary method of describing object interaction in object-oriented languages such as UML is via interaction diagram and sequence diagram. These diagrams provide very low-level description of messages passed between objects. The agent-oriented approaches provide richer description of agent interaction at a higher-level of interaction. Thus in \( i^* \) it is possible to describe how agents depend on each other.

### 2.3.1 The \( i^* \) framework

Understanding the organisational environment as well as the reasoning and rationale underlying requirements, design and process formulation decisions are crucial to model and build effective computing systems [161]. Conceptual modelling notations employing knowledge representation techniques have been developed to support such an understanding [158]. Many modelling techniques tend to address “late-phase” requirements while the vast majority of critical modelling decisions are arguably taken in early-phase requirements engineering [156][162]. The central concept in \( i^* \) is that of intentional actor. Intentional properties of an agent such as goals, beliefs, abilities and commitments are used in modelling requirements [157]. The actor or agent construct is used to identify the intentional characteristics represented as dependencies involving goals to be achieved, tasks to be performed, resources to be furnished or softgoals (optimisation objectives or preferences) to be satisficed. The \( i^* \) framework also supports the modelling of rationale by representing key internal intentional characteristics of actors/agents. The \( i^* \) framework consists of two modelling components [153]: Strategic Dependency (SD) Models and Strategic Rationale (SR) Models.

The SD and SR models are graphical representations that describe the world in a manner closer to the users perceptions. The SD model consists of a set of nodes and links. Each node represents an “actor”, and each link between the two actors indicates that one actor depends on the other for something in order that the former may attain some goal. The depending actor is known as depender, while the actor depended upon is known as the dependee. The object around which the dependency relationship centres is called the dependum. The SD model represents the goals, task, resource, and softgoal dependencies between actors/agents. In a goal-dependency, the depender depends on the dependee to bring about a certain state in the world. The dependee is given the freedom to choose how to achieve it. In a task-dependency, the depender depends on the dependee to carry out an activity. Task- and goal-dependencies may
often appear interchangeable. One way to understand the distinction is to view goals as more coarse-grained, abstract entities and tasks as more fine-grained, specific entities (while recognizing that goals can always be reformulated as tasks and vice versa). Another dimension to this distinction is the relative autonomy of the dependee in deciding how a goal is achieved, while in a task the depender and dependee must coordinate in a far more tightly-coupled fashion. In a resource-dependency, one actor (the depender) depends on the other (the dependee) for the availability of a resource. In each of the above kinds of dependencies, the depender becomes vulnerable in situations where the dependee fails to achieve a goal, perform a task or make a resource available. In a softgoal-dependency, a depender depends on the dependee to perform certain goals or task that would enhance the performance. The notion of a softgoal derives from the Non-Functional Requirements (NFR) framework [26, 27, 28] and is commonly used to represent optimisation objectives, preferences or specifications of desirable (but not necessarily essential) states of affairs.

As an example, consider a simplified version of the well-known Meeting Scheduler Scenario [156, 157]. This example will be used to illustrate the $i^*$ framework. The SD modelling process (see Figure 3.1) begins with identifying the actors involved with the meeting scheduling system and their mutual dependency relationships. The MeetingInitiator actor delegates much of the work of meeting scheduling to the MeetingScheduler. The MeetingInitiator no longer needs to be bothered with collecting availability information from participants, or to obtain agreements about proposed dates from them. The MeetingScheduler also determines what are the acceptable dates, given the availability information. The MeetingInitiator does not care how the MeetingScheduler does this, as longer as the acceptable dates are found. This is reflected in the goal dependency of MeetingBeScheduled from the MeetingInitiator to the MeetingScheduler actor. The MeetingScheduler expects the MeetingInitiator to enter the DateRange by following a specific procedure. This is modelled via a task dependency. Note that it is still the MeetingInitiator who depends on Participants to attend the meeting. It is the MeetingInitiator (not the MeetingScheduler) who has a stake in having Participants attend the meeting. The SD model models the meeting scheduling process in terms of intentional relationships among agents, instead of the flow of entities among activities. This allows analysis of opportunity and vulnerability. For example, the ability of a computer-based meeting scheduler to achieve the goal of MeetingBeScheduled represents an opportunity for the MeetingInitiator not to have to achieve this goal himself. On the other hand, the MeetingInitiator would become vulnerable to the failure of the
2.3. Agent-Oriented Conceptual Modelling

Figure 2.3: The Strategic Dependency Model of the Meeting Scheduling System

*MeetingScheduler* in achieving this goal.

An SR model (see Figure 3.2) represents the internal intentional characteristics of each actor/agent via task decomposition links and means-end links. The task decomposition links provide details on the tasks and the (hierarchically decomposed) sub-tasks to be performed by each actor/agent. The means-ends links in the SR model provides understanding about why an actor would engage in some tasks, pursue a goal, need a resource, or want a softgoal. From the softgoals, one can tell why one alternative may be chosen over others. The SR model thus provides a way of modelling stakeholder interests, and how they might be met, and the stakeholders evaluation of various alternatives with respect to their interests [156]. The SR model also provides constructs to model alternate ways to accomplish goals by asking “why”, “how” and “how-else” questions [159, 160].

For example, the *Participant* has an internal task to *ParticipateInMeeting*. This task can be performed by subtasks *AttendMeeting* and *ArrangeMeeting* (these are related to the parent task via task decomposition links). For the *MeetingInitiator*, the goal of *MeetingBeScheduled* is an internal goal. This goal can be met (represented via a means-ends link) by scheduling meetings in a certain way, consisting of (represented via task decomposition links): obtaining availability dates from participants, finding a suitable date (and time) slot, proposing a meeting date, and obtaining agreement
2.3. Agent-Oriented Conceptual Modelling

Figure 2.4: The Strategic Rationale Model of the Meeting Scheduling System

from the participants. In the case of Participant, the internal tasks FindAgreeableDateUsingScheduler and FindAgreeableDateByTalkingToInitiator are alternative means to achieve the goal Agreeable \((\text{Meeting, Date})\). How the alternatives contribute to softgoals is also represented. These are represented as means-ends link relationships. The SR model thus provides a way of modelling stakeholder interests, how they might be met, and the stakeholders’ evaluation of various alternatives with respect to their interests.

In order to develop the systems that meet the real needs of organisations, one should have a thorough understanding about the organisational environment and its impact on the system. For example, the analyst, before proceeding to improve the initial set of requirements, might do well to ask [156]:

- Why is it essential to schedule meetings ahead of time?
- Why does the meeting initiator want to ask participants for exclusion dates and preferred dates?
- Why is a computer-based meeting scheduler preferred? And whose interests does it serve?
- Is confirmation via the computer-based scheduler sufficient? If not, why not?
Most of the requirements modelling languages are not in a position to answer questions raised above. They tend to focus on the “what” rather than the “why”. Having answers to these “why” questions is important not only to help develop successful systems in the first instance, but also to facilitate the development of cooperation with other systems (e.g., project management systems, team coordination “groupware” for which meeting information may be applicable), as well as the ongoing evolution of these systems [156].

### 2.3.2 Agent-Object Relationship (AOR) modelling

Agent-Object-Relationship (AOR) modelling is another agent-oriented approach for the conceptual modelling of organisations and organisational information systems [144]. The crucial abstractions in the AOR notation are “entities” (i.e., agents, events, actions, commitments, as well as ordinary objects) and special relationships defined between them which are used to supplement the traditional association, aggregation/composition and generalization/specialization relationships of the UML models.

AOR modelling involves building both external and internal models. The external model aims at capturing the perspective of an external observer to model the domain producing one or more of the agents, interaction frames, interaction sequences, interaction pattern diagrams etc. The internal model aims at capturing the functionalities of the system for each focus agent. The internal AOR model is then refined for each focus agent into an implementation model for the target language/platform. AOR modelling language (AORML) tools as well as a Microsoft Visio template is available to support the AOR Modelling activities. AOR modelling covers requirements analysis, design, coding, and implementation phases (additional details about Agent-Object-Relationship (AOR) modelling can be found in [144]).

AOCM also finds application in other specialised settings of software engineering, such as web engineering (see the Journal of Web Engineering [41] and [21, 42, 43, 44] for further reference). A recent international workshop on Agent-based Web Engineering held in conjunction with the International Conference on Web Engineering 2005 (of which the author of this dissertation was co-chair) explored many of these issues.

### 2.4 Agent-Oriented Software Engineering

Recently, many complete agent-oriented software engineering methodologies have been proposed to model and design systems using agent-oriented concepts. The area of
agent-oriented methodologies is maturing rapidly and gaining wider acceptance [64]. In this section, we shall review some of the popular agent-oriented methodologies.

### 2.4.1 Prometheus

Prometheus [64, 107, 108, 109, 112, 149] is a comprehensive agent-oriented software engineering methodology. It provides support for system specification, architecture specification, detailed design and implementation. It has been applied in both industrial and academic environments. In Prometheus goals are specified using a textual notation combined with a simple graph notation to facilitate the decomposition of goals into sub-goals. Goals are identified from the initial description of the system and are refined into sub-goals using the procedure of asking “how” it is possible to accomplish the main goals. The methodology consists of three phases: system specification, architectural design, and detailed design [107].

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

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

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

The Prometheus methodology is supported by the Jack Development Environment (JDE) [2] as well as the Prometheus Design Tool (PDT) which can be used for the artifacts developed during the detailed design phase. Activities related to requirements elicitation are partially covered by the analysis phase (i.e., requirements analysis is explicitly included, though as part of the analysis phase). Additional details can be found in [107].
2.4.2 Gaia Methodology

The Gaia methodology [151, 164] can be used for the analysis and design of the agent-based systems. Gaia is a general methodology that is applicable to a wide range of multi-agent systems. The key concepts used in Gaia methodology are roles. Roles have responsibilities, permissions, activities, and protocols associated with them. Roles can interact with one another in certain institutionalised ways, which are defined in the protocols of the respective roles. The Gaia methodology covers a limited number of phases in the design process, namely analysis and design (the latter includes the architectural and detailed design). Requirements elicitation, coding and implementation, verification and testing, and deployment, are not considered.

The Gaia methodology design process consists of three types of models: agent model, services model and acquaintance model [151]. The agent model discovers the agent types that will make up the system, and the agent instances that will be instantiated from these types. The services model is used to identify the main services that are required to realise the agent’s role. The acquaintance model documents the lines of communication between the different agents. There are three steps involved in the Gaia methodology’s design process. The first step is used to map the roles into agent types and to create the right number of agent instances of each type. The second step is used to find out the services model that will be needed to fulfill a role in one or more agents. The third step is used to create the acquaintance model for the representation of communication between different agents. Additional details can be found in [151].

2.4.3 Other Approaches

The Roadmap [79, 80] methodology extends the Gaia methodology [151] by introducing use cases, a dynamic role hierarchy and additional models to illustrate the agent environment and agent knowledge. In the present form, UML is sometimes inadequate for modelling agents and agent-based systems. The AUML effort [6, 104] started from UML and extended it with additional abstractions and notations specifically tuned to the problems of agent-based applications. AUML has achieved good results in modelling complex interaction protocols between components. Roadmap also introduces an interaction model based on AUML interaction diagrams. Roadmap mainly focuses on the analysis and design phases.

The MaSE methodology [38] is structured in seven steps. These steps are concerned
with capturing goals, applying use cases, refining goals into roles and capturing their interactions, creating agent classes, constructing conversation, assembling agent classes, and designing the system [38]. MaSE is supported by a tool known as agentTool [150]. The compositional multi-agent design method DESIRE (DEsign and Specification of Interacting Reasoning components) [15] supports the design of component based on autonomous interactive agents. This design is based on conceptual design. The system’s specification takes the advantages of knowledge-based techniques.

The Tropos methodology [24] will be discussed in the next section.

### 2.5 Early- and Late-phase RE

Recent proposals have made a distinction between early-phase and late-phase Requirements Engineering [23, 156]. Most of the proposed late-phase RE techniques usually focus on completeness, consistency, and automated verification of requirements [156]. These techniques deal with the production of a requirement document such that the resulting system would be sufficiently specified and constrained in a contractual setting. On the other hand, a different kind of support is necessary for the early phase of RE. Some common questions that need to be addressed in the early-phase are the following [156]:

- What are the main goals of the system?
- Why do stakeholders depend on each other?
- What alternatives exist?

The $i^*$ framework [154, 155, 158, 161, 162, 163] is an Agent-Oriented Conceptual Modelling (AOCM) language (agent- and goal-oriented modelling language) that has been specifically designed for early-phase requirements engineering and that is well-suited for answering questions such as these. The emphasis here is on understanding the motivation and rationale that underlie system requirements (“Why’s” than “What”). This, in turn, supports a more refined analysis of system dependencies and encourages a uniform treatment of the systems functional and non-functional requirements.

The transition from early to late phase requirements constitutes a challenging problem [156]. This is based on the fact that frequent concept mismatch occurs and a developer might make use of domain knowledge partly expressed in descriptions of the organisation, and partly in the existing requirements specifications. Few attempts
have been made to address the integration of early and late requirements and as a result we end up with systems being developed that fail to support critical organisational goals, stakeholders intentions and rationale [145]. If we propose to produce high quality systems, it is vital to try to bridge this gap.

Several proposals exist for integrating $i^*$ modelling with late-phase requirements analysis and the downstream stages of the software life-cycle. The Tropos methodology [24] explores how $i^*$ models might be refined to form the basis for late-phase requirements specifications, and subsequently architecture and detailed design specifications. The $i^*$ notation alone is not adequate for representing the level of detail necessary for late-phase requirements specifications. To address this, formal approaches such as Formal Tropos [55] have been developed. Alternative approaches has been proposed to define methodologies for transforming $i^*$ models into agent programs in formal agent programming languages such as ConGolog [145], AgentSpeak(L) [125] and 3APL [61].

Before proceeding to discuss other approaches, we shall explain the Tropos methodology [24].

2.5.1 Tropos Methodology

Tropos is an agent-oriented software engineering methodology that can be used to represent early and late requirements, as well as architectural and detailed design. Tropos is requirements-driven in the sense that it is based on concepts used during early-phase requirements analysis. It adopts the concepts offered by $i^*$ [156, 157], and is suitable for the design of agent-oriented, distributed, open, applications.

Two key ideas underpin the Tropos methodology. First, the notion of agent and related mentalistic notions such as the goals and plans, are used in all the phases of software development, from early analysis through to the actual implementation. For example, mentalistic notion of goal is achieved when one agent depends on the other to bring about certain state in the world [156]. Second, Tropos covers all phases of software development. The early requirements phase is concerned with the requirements engineer identifying the domain stakeholders and modelling them as social actors, who depend on one another for goals to be achieved, plans to be executed, and resources to be furnished. Based on these dependencies, it is then possible to reason about the “why” and “how” of the system functionalities. The other goal here is to verify how the final implementation matches initial needs. In the late requirements phase, this model is extended including a new actor, which represents the system, and a number of dependencies with other actors in the environment. These dependencies are then used
in defining all the functional and non-functional requirements of the system-to-be. The focus of the other two phases, architectural design and the detailed design phases is on the system specification, based on the requirements resulting from the requirements phases. Architectural design is defined in terms of subsystems interconnected through data and control. The Detailed Design phase aims at specifying agent capabilities and interactions. Implementation is agent-based, in the JACK agent development environment. Due to its reliance on rich early-phase (agent-oriented) models of organisational context, the Tropos framework allows for a deeper understanding of the environment where the software-to-be is going to eventually operate [24].

2.5.2 Formal Tropos

Formal Tropos (FT) [55] is part of the Tropos project [24]. FT aims to achieve an effective integration of Formal Methods in the Tropos software development process. An FT specification consists of a sequence of class declarations for entities, actors, goals, and dependencies. Each of these declarations associates a set of attributes to the class and characterises its instances [55]. Additionally, these class declarations contain temporal constraints expressed in a typed first-order linear time temporal logic (LTL) (LTL is also used in the KAOS methodology [33]). LTL has been renowned as leading technique for the specification of temporal rules/constraints [55]. These constraints are used to describe the valid lifetime evolutions of the model in terms of temporal evolutions of set of instances of the classes in the specification. Two critical moments specified in the life-cycle of goals and dependencies are the instants of their creation and fulfillment conditions. The creation condition of a goal is interpreted as the moment in which the depender expects or desires to achieve the goal, while its fulfillment is the moment in which the goal condition is actually achieved. Creation and fulfillment constraints in the FT specifications can be used to define conditions for these two moments in the life of intentional elements. For defining constraints on the lifetimes of sub-goals in a goal decomposition, creation and fulfillment conditions are used (based on the notion that sub-goals are created after the parent goal and should be fulfilled before the parent goal can be fulfilled), or for defining the responsiveness of an actor based on the dependencies (an actor can take care immediately of some of the dependencies while delaying other dependencies). FT also has the provision to introduce invariant constraints that are used to define conditions that should be true throughout the lifetime of a class instances. Typically, invariants define relations on the possible values of attributes, or cardinality constraints on the instances of a given
Figure 2.5: Fundamental actors of the case study and their goals [55]

class [55].

The SD model of the Insurance Company case study shown in Figure 2.6 will be used to illustrate FT specifications [55]. The identified actors in the case study are the customers and the insurance company, Customer and InsuranceCo. The main identified goal of the customer is to be compensated for any damages in the case of an accident (goal shown as BeInsured in Figure 2.5). Customer is not in a position to fulfill this goal by herself/himself, but depends on the insurance company for the realisation of goal dependency CoverDamages. Entities in FT specifications are used to represent non-intentional elements that exist in the environment or organisational context, but they should not be directly related to an actor’s strategic goals. In the current example (refer Figure 2.6), we have the entities Claim and Car. In the following, we provide an FT specification of these two entities and the two dependencies CoverDamages and RepairCar. An attribute in the FT class denotes the relationships among the different objects being modeled. For example, every claim made by a customer refers to a particular car, represented by attribute car of entity Claim. Actors in FT (same as i*) may have goals that may cater to their strategic interests. Actor Customer has a goal BeInsured. Also, all the intentional relationships between actors are represented as dependencies (for e.g., dependencies CoverDamages and RepairCar as an example). The inner layer in the FT class declaration describes the constraints on the possible evolution of the instances of that class [55]. The example we have taken states that the car should not be working at the time the goal dependency CoverDamages is created. The creation condition represents that whenever a customer fills a claim then a dependency for covering the repairs is created with the customer as depender and insurance company as dependee. Also represented in the specifications is the goal
dependency of repairing a car. This can only arise if the car is not working. Similarly, a necessary condition for the fulfillment of the dependency is that the car should be running OK.

Dependencies are characterised by mode, which declares the attitude of the actors with respect to its fulfilment [55]. Four modalities are currently supported by FT. In the achieve mode, fulfilment properties should be satisfied at least once. In the maintain mode, fulfilment properties should be satisfied in a continuing way. achieve&maintain is the combination of the previous two modes. In the last mode avoid, the fulfilment properties should be avoided.

**Entity** Claim

**Attribute constant** car: Car

**Entity** Car

**Attribute** runsOK: boolean
2.5. Early- and Late-phase RE

**Actor** InsuranceCo

**Actor** BodyShop

**Actor** Customer

  **Goal** BeInsured

  **Mode** Maintain

  **Fulfillment definition**\( \forall \text{cov} : \text{CoverDamages} \)

  \((\text{cov}.\text{depender} = \text{self} \rightarrow \Diamond \text{Fulfilled(cov)})\)

**Dependency** CoverDamages

  **Type goal**

  **Mode** achieve

  **Depender** Customer

  **Dependee** InsuranceCo

  **Attribute constant** cl: Claim

  **Creation**

  **condition** \( \neg \text{cl.car.runsOK}\)

  **trigger** JustCreated(cl)

**Dependency** RepairCar

  **Type goal**

  **Mode** achieve

  **Depender** Customer

  **Dependee** BodyShop

  **Attribute constant** cl: Claim

  **Creation**

  **condition** \( \neg \text{cl.car.runsOK}\)

  **Fulfillment**

  **condition for depender** cl.car.runsOK

FT specifications can also be used to specify properties that should hold in the domain, so that they can be verified against the model. FT distinguishes between assertion properties that should hold for all the valid evolutions of the FT specification, and possibility properties that should hold for at least one. Provided the FT specification and a set of properties, the analyst will be in a position to verify whether the FT specification satisfies the properties by means of T-TOOL [55], an automatic verification tool based on the NuSMV [29] model checker (refer to [54] for more details). The usefulness of the Tropos (and of the FT) methodology has been illustrated by several
2.5. Early- and Late-phase RE

2.5.3 Combined use of \(i^*\) and ConGolog

An alternative approach has been proposed to define methodologies for transforming \(i^*\) models into agent programs in formal agent programming languages such as ConGolog [145, 37]. The proposed methodology makes use of the \(i^*\) framework to carry out early-phase RE, that is, model and analyse intentional relationships between actors, reason about the rationale behind their activities, vulnerabilities and opportunities for actors, and evaluate different alternatives for the process. This methodology uses an intermediate notation, known as annotated SR diagrams, to specify processes in a precise fashion so that they can be mapped into the ConGolog framework. The methodology also involves using the ConGolog model of the process to validate the specification by performing simulation experiments. This can be used to evaluate whether the process proceeds as per the analyst’s expectations.

This methodology mostly relies on a set of annotations that are introduced into the \(i^*\) SR diagram to allow more detailed information about the processes to be represented. Two types of annotations are used: Link and Composition annotations. Link annotations are used to state under what conditions a task/goal should be performed, and whether it should be performed repeatedly. Composition annotations are used to specify whether the subtasks/subgoals associated with a decomposition link should be performed concurrently, sequentially, concurrently with different priorities, or whether they are alternatives. These annotations are used to allow the analyst to specify the details of how a process should proceed and help him/her and the client to clarify their system specification choices. Guidelines for generating an annotated \(i^*\) SR diagrams are provided. Softgoals and their related links are suppressed in the initial analysis, which involves, amongst others, the operationalisation of dependencies to clarify the communication behavior of the different actors to achieve a goal or perform a task or provide a resource. Decomposition links are also annotated using the defined link notations and composition annotations. This produces a precise specification of the processes involved in the system. The analyst is required to define a mapping from the components of his/her annotated SR diagram into the components of a ConGolog model using some mapping rules. An additional set of mapping rules is defined to help an analyst maintain consistency between the \(i^*\) and ConGolog models [145]. The mapping rules constrain the modeler to map elements of the annotated SR diagram
into appropriate entities in the ConGolog model and ensure that the models are consistent. This allows traceability between corresponding elements in the two models when changes are made. Every element in the annotated SR diagram must be mapped into an appropriate element of the ConGolog model. If some part of the annotated SR diagram needs to be changed, then the corresponding part of the ConGolog model can be updated too, and vice versa. The mapping rules provide formal semantics for annotated SR diagrams by reducing them to ConGolog, which has well-defined formal semantics [145].

### 2.5.4 Combined use of $i^*$ and Albert-II

ALBERT-II [46] specifies requirements formally through states, actions, information and perception. It is designed for the purpose of modelling functional requirements connected with distributed heterogeneous real-time cooperative systems [166, 47]. The design of the language focuses on three aspects: agent-orientation, formality and expressiveness. Bissener has studied the combined use of the $i^*$ and Albert-II frameworks for requirements engineering in [9]. He proposed a methodology that deals with the organisational, non-functional, as well as functional requirements. The functional requirements are specified in Albert-II and the organisational issues (understanding and redesigning organisational process through strategic relationships and rationales) and non-functional requirements are specified in $i^*$.

### 2.5.5 Combined use of $i^*$ with 3APL and AgentSpeak(L)

Recently, two approaches have been proposed to define methodologies for transforming $i^*$ models into agent programs in reactive, BDI (Belief Desire and Intention) agent programming languages such as 3APL[61] and AgentSpeak(L) [125]. These approaches are based on executing $i^*$ models by translating them into a set of interacting agents implemented in the 3APL and AgentSpeak (L) languages respectively. The proposed approaches make use of the advantages of $i^*$ for modelling early-phase requirements and help validate the model by mapping it into an executable agent-based specification [61]. The facility to execute an $i^*$ model via multi-agent system permits the verification of an extensive range of business rules, constraints and conditions such as creation conditions, invariant conditions and fulfillment conditions of the kind used in the Formal Tropos. The starting points and motivations for both exercises are similar, but the eventual mapping of models to multi-agent systems is defined in a very different manner.
Table 2.1: Scope of the discussed Approach/Methodology

<table>
<thead>
<tr>
<th>Approach/Methodology</th>
<th>KAOS</th>
<th>NFR</th>
<th>i*-ConGOLOG</th>
<th>Tropos</th>
<th>i*-3APL, AgentSpeak(L)</th>
<th>i*-Albert-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Early-Phase RE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Late-Phase RE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Architectural Design</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Detailed Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The main advantages of using 3APL over AgentSpeak(L) are, 3APL uses the notion of goal rather than the notions of event and intention and it has a larger range of rules which enable agents to modify, revise, skip or drop goals when there are failure [61]. On the other hand, the main advantages of using AgentSpeak(L) over 3APL are, the models underlying the 3APL assume a time-sliced execution of agents that is inefficient when agents are inactive most of the time. The AgentSpeak(L) in contrast supports event-based reactive behaviour and pro-active (goal directed) behaviour. The AgentSpeak(L) is widely accepted with a solid theoretical and philosophical foundation [14, 115, 116, 117, 118, 119]. Finally, there is a evidence to suggest that systems based on the AgentSpeak(L) can be used to build successful applications [65].

Table 2.1 summarises the scope of the approaches presented. The identified phases are based on the Tropos methodology.

2.6 Formal Methods

Formal methods are system design techniques that use strictly specified mathematical models to build software and hardware systems. In comparison to other design systems, formal methods use mathematical proof as a complement to system testing in order to ensure correct behavior. Wide ranging definitions of formal methods may be found in the literature. According to Jeannete Wing [148], most formal methods are defined in terms of a specification language that has a well-defined logical inference system. A logical inference system defines a consequence relation, typically given in terms of a set of inference rules, mapping a set of well-formed sentences in the specification language to a set of a well-formed sentences. We use this inference system to prove properties
from the specification about specificands. Viewing a specification as a set of facts, we derive new facts through the application of the inference rules.

When a statement can be inferred from a set of facts, we are able to prove a property that a specificand satisfying the specification will have, a property not explicitly stated in the specification. An inference system provides the users of formal methods a way to predict a system’s behavior without having to execute or even build it. It gives users a way to state questions, in the form of conjectures, about a system cast in terms of just the specification itself. Users can then answer these questions in terms of a formal proof constructed through a formal derivation process. The inference system increases user confidence in the specification’s validity. If users are able to prove a surprising or undesirable result from the specification, this might point to errors in the specification. A formal method with an explicitly defined inference system usually has the further advantage that this system can be completely mechanized (e.g. if it has a finite set of finite rules) [148]. Theorem provers and proof checkers are examples of tools that assist users with deriving and managing formal proofs.

A major advantage of a formal notation is that it is precise and unambiguous. Thus the formal notation always provides the definitive description in the case of any misunderstanding. Formal methods can benefit any stage of the development lifecycle. The strictness and clarity necessary for its effective use opens the way for more effective requirements capture and analysis, cleaner and more robust architectures and designs, provably correct software, better documentation, enhanced maintainability [13].

Formal methods can be used selectively at any stage in the development lifecycle where its benefits can be evidently justified. For example, a safety-critical system development might need fully formal proofs at each stage, whereas a less critical application might benefit mainly from just a formal requirements model, followed by a further conventional software development process.

A lightweight formal method is a recently proposed approach that combines classical notions of formal specification and verification with new automatic checking technologies, and a more cost-effective and risk-driven style [67]. Alloy [67, 68, 69, 70, 71, 72, 82] is a lightweight modelling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer, and offers a visualizer for providing sense of the possible solutions and counterexamples.

Formal languages such as Z [13, 133, 134], VDM [76], B [1] are widely accepted and popular in industry and theoretical investigations. The next subsection introduces a general specification language, Z (’zed’).
2.6.1 The Z notation

We discuss the combined use of i* models and Z specifications in chapter’s 4 and 5. This section provides the reader with a quick introduction to Z.

This introductory description is based on [13, 20, 133, 134], which the reader may refer to for additional details. Z has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG) within the Computing Laboratory [62, 111, 133, 134]. It is a typed language based on set theory and first order predicate logic. The notation is useful (once mastered) to organise the thoughts and assist the communication of ideas within a design team. It is also readable enough to be used as a documentation tool in a manual. Natural language should also be included to provide an informal description of the system and to relate the mathematical descriptions to the real world.

Z is supported by comprehensive tool support. First of them is CADiZ [77], which is a UNIX based tool for checking and typesetting Z specifications. Zola the WYSIWYG editor is another interesting tool, which supports the production and typesetting of Z specifications. Also included are a type-checker and a Tactical Proof System (available from http://www.ist.co.uk/PRODUCTS/zola.html). Z type checkers like ZTC [74] and Z animation tools like ZANS [75] can be used to analyse Z specifications. It is projected to be compliant with the second edition of Spivey’s Z reference manual [132]. Formaliser [53] is a syntax-directed Z editor as well as an interactive type-checker, running under Microsoft Windows obtainable from Logica. Z/EVES [106] supports the analysis of Z specifications in several ways: for syntax and type checking, schema expansion, and precondition calculation.

The idea behind an abstract Z specification is to describe what a system does rather than how it does it. Imperative programming languages are specifications, but these concentrate on how the result is to be achieved. Functional programming languages are more like specification languages since these describe what result is required. However they are designed to be executable. Z can be used in a functional style. However it is possible (and sometimes desirable) to write non-deterministic specifications in Z. This means the exact execution of the specification cannot be determined. The Z notation is designed to be expressive and understandable (by humans) rather than to be executable. Even though a Z specification is not in general executable, by passing it around members of a design team it may be mentally executed and checked far more reliably than an equivalent informal specification [13].

Z has found acceptance in industry [13]. Many methods are imposed on designers
in industry by managers attempting to improve efficiency. From the feedback which has been obtained, it seems that the use of Z is one of the few specification techniques which has not been received with reluctance by industrial users [13]. The notation is gradually gaining acceptance in industry, and is taught in many computer science curricula [13].

Z includes a schema notation to aid the structuring of specifications. It has been found convenient to use a state or model based approach in representing Z schemas [13]. A system may be considered to be modelled as an abstract state and a sequence of operations on this state. In other words, schemas are primarily used to specify state spaces and operations for the mathematical modelling of the systems [13]. For example, the following is a schema called StateSpace:

\[
\text{StateSpace} \\
x_1 : S_1; \ldots; x_n : S_n \\
\text{Inv}(x_1, \ldots, x_n)
\]

This schema denotes a state space in which \( x_1, \ldots, x_n \) are known as state variables and \( S_1, \ldots, S_n \) are expressions from which their types may be systematically derived. Sets \( x_1, \ldots, x_n \) should not occur free in \( S_1, \ldots, S_n \). If they do occur, they refer to other occurrences of these variables already in scope (e.g., globally defined variables). \( \text{Inv}(x_1, \ldots, x_n) \) is the state invariant, relating the variables in some way to all possible allowed states of the system during its lifetime. In other words, the state variables will also have some state invariants associated with them representing “healthiness conditions” which must always be satisfied.

For example, the state variables of a counter system may be specified using the following schema [20].

\[
\text{Counter} \\
\text{ctr} : N \\
0 \leq \text{ctr} \leq \text{max}
\]

Here, \( \text{ctr} \) is declared to be a natural number and the predicate part describes an invariant that must be satisfied by \( \text{ctr} \), the state variable of the system.

An initialisation may be specified as follows [20]:

\[
\text{InitCounter} \\
\text{Counter} \\
\text{ctr} = 0
\]
An operation is specified in Z with a predicate relating the state before and after the
invocation of that operation [20]. For example, an operation to increment the counter
may be specified as follows:

\[
\begin{align*}
\text{Increment} \\
\Delta \text{Counter} \\
\text{ctr} < \text{max} \\
\text{ctr}' = \text{ctr} + 1
\end{align*}
\]

The declaration \( \Delta \text{Counter} \) means that the state Counter is changed by the opera-
tion. In the predicate, the new value of a variable is primed (\( \text{ctr}' \)), while the old value
is unprimed. So the above predicate states that the new value of the counter, \( \text{ctr}' \), is
the old value plus one. Note that there is an implicit conjunction (logical-and) between
successive lines of the predicate part of a schema.

As well as changing the state variables, an operation may also have input and out-
put parameters. Input parameter names are usually suffixed with ‘?’ , while output
parameter names are suffixed with ‘!’ . For example, the following operation for decre-
menting the counter has as an input parameter, the amount by which the counter
should be decremented [20] :

\[
\begin{align*}
\text{Decrement} \\
\Delta \text{Counter} \\
d? : \mathbb{N} \\
\text{ctr} \geq d? \\
\text{ctr}' = \text{ctr} - d?
\end{align*}
\]

The next operation has an output parameter, which is the value of the counter.
The declaration \( \Xi \text{Counter} \) shows that the operation cannot change the state of the
Counter [20].

\[
\begin{align*}
\text{Display} \\
\Xi \text{Counter} \\
c! : \mathbb{N} \\
c! = \text{ctr}
\end{align*}
\]

Types are used to differentiate the various forms of data present in a specification
[20]. Advantages of using types are that they

- Help to structure specifications by differentiating objects
• Help to prevent errors by not allowing us to write meaningless things
• They can be checked via automated tools.

The declaration \( x : T \) shows that \( x \) is of type \( T \), where \( T \) is a set. This is like saying \( x \in T \). A new basic type \( T \) is introduced to a specification by putting its name in square brackets:

\[
[T]
\]

This allows us to name the types of a specification without saying what kind of objects they contain. For example, a specification of an address book might introduce the basic types \( \text{Name} \) and \( \text{Address} \) without worrying about the structure of these types:

\[
[\text{Name}, \text{Address}]
\]

If we know the exact values of a type we use an enumerated type declaration:

\[
\text{Direction ::= north | south | west | east}
\]

Sets have types too. The type of the set \( \{1, 4, 5, 8\} \) is “set of \( N \)”. More precisely, this is written:

\[
\{1, 4, 5, 8\} : \mathbb{P}N
\]

Predicates are used to state truth properties of values in a specification. Examples of simple predicates include:

• \( (z+2) = 7, \ z \in T \)

Compound predicates are formed using the following logical operators:

• And \( M \land N \) (Conjunction)
• Or \( M \lor N \) (Disjunction)
• Implies \( M \rightarrow N \)

Universal Quantification is written as follows:

\[
\forall X : T \bullet A
\]

Existential Quantification is written as follows:

\[
\exists X : T \bullet A
\]

Consider the example of a specification of a system used to check staff members in and out of a building. This discussion on Z notation is based on [20]. Since we shall be dealing with elements of type staff, we introduce the type \( \text{Staff} \) as a basic type:
2.6. Formal Methods

[Staff]

The state of the system is described by the following schema;

\[
\begin{align*}
\text{Log} & \\
\text{users, in, out} & : \mathcal{P}\text{ Staff} \\
\text{in} \cap \text{out} & = \emptyset \land \\
\text{in} \cup \text{out} & = \text{users}
\end{align*}
\]

The state consists of three components modelling

- the set of users of the system
- the set of staff members who are currently in and
- the set of staff members who are currently out

The predicate part of the state schema describes an invariant of the system. The invariant says that

- No staff member is simultaneously in and out.
- The set of users of the system is exactly the union of those who are in and those who are out.

An operation to check a staff member into the building is specified as follows:

\[
\begin{align*}
\text{CheckIn} & \\
\Delta \text{Log} & \\
\text{name?} & : \text{Staff} \\
\text{name?} & \in \text{out} \land \\
\text{in}' & = \text{in} \cup \{\text{name?}\} \\
\text{out}' & = \text{out} \setminus \{\text{name?}\} \\
\text{users}' & = \text{users}
\end{align*}
\]

This has an input parameter representing the member of staff to be checked in. The predicate part says that:

- The staff member to be checked in must currently be out. This is a pre-condition on the operation.
- The staff member is added to the set in.
• The staff member is removed from the set out.

• The overall set of users remains unchanged.

Similarly, an operation to check a staff member out of the building may be specified as follows:

<table>
<thead>
<tr>
<th>CheckOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>△Log</td>
</tr>
<tr>
<td>name? : Staff</td>
</tr>
</tbody>
</table>

| name? ∈ in          |
| out′ = out ∪ \{name?\} |
| in′ = in \{name?\}  |
| users′ = users      |

2.7 Unified Modeling Language (UML)

This brief introduction to UML and UML sequence diagrams in this section is intended to provide the reader with sufficient background material to understand our discussion of the combined use of i* and UML sequence diagram in chapter 6.

UML represents the combination of three earlier influential object-oriented modelling languages [122]. Booch, Rumbaugh and Jacobson were the authors of these three languages. UML went through a standardisation process with the Object Management Group (OMG), and it is now an OMG standard [136]. UML’s semantics are based on the object-oriented paradigm (OOP) [3]. The OOP is based on the concepts such as classes, associations, objects and links, attributes, operations and methods, messages and stimuli. The OOP is based on four principles that permit us to manage change and complexity: Abstractions, Encapsulation, Generalisation and Polymorphism. Additional details about OOP can be found in [3, 122].

The present UML notational standard addresses the system analysis, design, and deployment steps in a development lifecycle. A UML-based development approach usually involves a series of graphical models which are used to define the functional and technical aspects of an application system. Each model depicts a diagrammatic representation of one aspect of the application and is combined with the other model objects. These models are then used as the component specifications for the construction phase of the project.

Some of the advantages of using UML are based on the fact that it is a common language that can be used from the conception of the product to delivery. The use of
UML considerably reduces the learning curve across projects. It supports the notion of increased domain and design model reuse and increased customer involvement during the project. Also, the wider acceptance of UML in industry is due to the fact that it is basically a visual modelling language, which helps software developers to visualise the software from multiple dimensions. This helps in understanding the future system (being developed) before the actual development work begins. Additionally, UML can be used to create numerous models at increasing levels of detail.

The UML has a number of models which in turn form part of the models for systems development, such as the systems requirements [88]. We shall provide the flavour of the models available in UML. Use cases define exactly the points at which a computer system is used. Use cases define interactions between the users of the system and the system itself. To support the description of use cases, some additional notations and conventions has been introduced. Activity diagrams describe activities and the flow between activities. These diagrams can be used in the systems analysis to specify the behaviour of use cases. Statechart diagrams are alternative form of diagrams that are mostly used at the design stage. They permit description of some object as it moves between states. These diagrams allow the analysts and designers to work at varying level of details. A sequence diagram is a key design diagram in UML. It is extensively used in the design process [3]. It can be used to identify interactions, as well as basis for determining what objects are needed in the system to provide the functionality. Collaboration diagrams are similar to sequence diagrams, but they represent interactions in a different ways. They may show different aspects of the activity but overlap in detail. Class diagrams define the static and unchanging structure of the system. The class diagrams represent the paths for all possible use of classes. They provide information about the data that is to be stored in the system and the details of the processing that takes place. Component diagrams are used to group together the elements of a computer system into larger units. Inside each component there will be a number of objects, defined elsewhere in the class diagrams. The components are then grouped together to cooperate to provide a complete system [88]. A complete coverage of UML can be obtained from [3, 88, 122, 136].

The most important concepts in understanding UML are: the UML architecture, notation (or diagrams), constraints and extension mechanisms. It consists of a set of symbols (the notation) and a group of rules (semantics) that manage the language. Rules in UML may be classified into:

- Syntactic: used to specify the aspect and combination of rules
• Semantic: used to specify the meaning of the symbols, individually and in context
• Pragmatic: provide the guidelines on how to use the language (the intent of the symbols).

OMG has now released UML 2.0. The main driver for the latest version is OMG’s objective of making UML capable of supporting the Model Driven Architecture (MDA) approach, which meant that the UML had to function better as a model driven notation. Also, the UML 1.x notation set was at times difficult to apply to larger applications. Furthermore, the notation elements needed to be improved in order to make diagrams more readable. (For example, modelling logical flow in UML 1.x was complicated and at times impossible. Changes to the sequence diagram’s notation set in UML 2.0 have made vast improvements in modelling logic in sequences)[7]. The next subsection introduces a general introduction to UML Sequence diagram (one of the nine diagrams used in UML), which is directly related to this work.

### 2.7.1 UML Sequence Diagram

A UML sequence diagram is a dynamic model that illustrates the interaction between objects arranged in a time sequence [40]. These diagrams are widely used both in theoretical investigations and in practice [122]. Sequence diagrams can be drawn at different levels of detail and to meet different purposes at several stages in the development life cycle. The most widely used application of a sequence diagram is to represent the detailed object interaction that occurs for one use case or for one operation. When a sequence diagram is used to model the dynamic behaviour of a use case it can be seen as a detailed specification of the use case. Sequence diagrams can also be used to document how objects in an existing (‘legacy’) system currently interact [7].

A sequence diagram has two dimensions [136]:

• The vertical dimension shows the sequence of messages/calls in the temporal order that they occur

• The horizontal dimension shows the object instances from which messages originate and to which messages are sent.

Each sequence diagram is represented by a notation element called a frame. This is the graphical boundary of the sequence diagram. The diagram’s label is placed on the top left hand corner of the sequence diagram frame. A scenario consists of a temporal sequence of interaction events between the (future) system and its environment.
Scenarios form real system descriptions from a usage-oriented perspective. They focus on aspects of reality describing particular cases of the system’s usage (i.e. its direct or indirect relation to users of the system), its relation to other systems and its relation to the real world. After making this knowledge explicit scenarios help analysts in reasoning about complex systems. Various sequence diagrams (for different scenarios) can be used to detect bottlenecks within the system design. By paying attention to what messages are being communicated between the objects, and looking at how long it takes to run the invoked method, analysts can quickly understand where the design has to be changed to distribute the load within the system.

During the modelling process of the sequence diagrams, lifeline notation elements are placed across the top of the diagram. Lifelines are used to represent either roles or object instances that participate in the sequence being modelled. Lifelines are drawn as a box with a dashed line descending from the center of the bottom edge. The lifeline’s name is placed inside the box. Messages are represented as horizontal arrows between the lifelines. These arrows contain text that describes a message being passed from one actor to another. A message is a basic form of communication in a sequence diagram interaction. The message is used to specify the kind of communication as well as information about the sender and the receiver.

In a sequence diagram, we can show how the system or object may call its own methods. To model an object calling itself, connection of the message back to the object itself is realised. “Alternatives” are used in sequence diagrams to select a mutually exclusive option between two or more message sequences. The word “alt” is placed inside the frame’s namebox.

In the example shown in Figure 2.7 (based on [7]), the analyst object makes a call to the system object which is an instance of the ReportingSystem class. The analyst object is calling the system object’s getAvailableReports method. The system object then calls the getSecurityClearance method with the argument userId on the secSystem object, which is of the class type SecuritySystem.

In addition to just showing message calls on the sequence diagram, the diagram may include return messages. These return messages are optional. A return message is drawn as a dotted line with an open arrowhead back to the originating lifeline, and above this dotted line can be placed the return value from the operation. The use of return messages depends on the level of detail/abstraction that is being modelled. Return messages are useful if finer detail is required. Otherwise, the invocation message is sufficient. In Figure 2.7 the secSystem object returns userClearance to the system.
When modelling a sequence diagram, there will be times when an object will need to send a message to itself. A purist would argue that an object should never send a message to itself. However, modelling an object sending a message to itself can be useful in some cases. For example, Figure 2.8 (based on [7] is an improved version of Figure 2.7. Figure 2.8 shows the system object calling its determineAvailableReports method. By showing the system sending itself the message “determineAvailableReports”, the model draws attention to the fact that this processing takes place in the system object.
2.8 Summary

In this chapter, we have given a brief introduction of the major areas related to our work. Those areas are Requirements Engineering, Goal-oriented RE, Agent-Oriented Conceptual Modelling, Agent-Oriented Software Engineering, Early and Late-Phase RE, Formal Methods and UML (including UML Sequence Diagrams).
Chapter 3

Requirements elicitation for Agent-Oriented Conceptual Modelling

One aim of this thesis is to present a novel requirements elicitation technique for the $i^*$ notation. This chapter describes this technique. A simplified computer based training (CBT) system for volunteers Case Study (this is part of collaborative project to build a comprehensive enterprise model for an emergency services agency) is used to illustrate the technique. We then introduce the notion of ontology-driven requirements elicitation and argue for an elicitation methodology that involves co-evolution of conceptual models, ontologies and requirements capture templates.

3.1 Introduction

Understanding the organisational environment as well as the reasoning and rationale underlying requirements, design and process formulation decisions is crucial to model and build effective computing systems [161]. Agent-Oriented Conceptual Modelling (AOCM) offers an interesting approach in modelling the early phase requirements. Given the relative infancy of agent-oriented conceptual modelling notations (including $i^*$), few attempts have been made to deploy them in industry-scale settings. This chapter presents some methodological principles and lessons derived from a collaborative project to build a comprehensive enterprise model for an emergency services agency (this is also one of the earliest attempts at industry-scale deployment of agent-oriented conceptual modelling techniques). The emergency services agency offers some unique features, such as an infrastructure that remains dormant for long periods of time but gets activated during an emergency. These features make traditional conceptual modelling somewhat difficult to use, but renders the domain eminently suitable for the deployment of agent-oriented conceptual modelling techniques.

Our focus in this chapter is on techniques to support requirements elicitation in an AOCM context. We begin by explaining the $i^*$ notation using (simplified) examples
3.2 SD Model of the Computer Based Training System (CBT) for Volunteers

drawn from the real-life conceptual modelling exercise. We present two specific outcomes of our experience with early-phase requirements engineering in the emergency services domain. First, we show how it is possible to use the underlying modelling notation to drive the design of a set of Requirements Capture Templates that can simplify the process of requirements elicitation via stakeholder interaction. In effect, these templates are forms that an analyst/modeller seeks to complete in the course of a stakeholder interview which provide structure and direction to an otherwise unstructured exercise. Once completed, these templates can be (manually) transformed in a relatively straightforward fashion to eventual Strategic Dependency (SD) and Strategic Rationale (SR) models. In addition, these completed templates can serve as a structured repository and record of stakeholder interaction that can be revisited whenever requirements need to be re-negotiated or revised (for instance, to resolve an inconsistency) or business processes need to be re-engineered. Second, we argue for a direct role for enterprise ontologies to support the elicitation process. We outline an approach where domain ontologies (specific to the relevant application domain) can help generate elicitation triggers. We show how completeness and consistency testing of conceptual models relative to domain ontology can suggest elicitation probes to the analyst/modeller. We then argue for a methodology that supports the co-evolution of conceptual models, ontologies and requirements capture templates through the process of elicitation and modelling.

3.2 SD Model of the Computer Based Training System (CBT) for Volunteers

The SD model provides an important level of abstraction for describing systems in relation to their environments, in terms of intentional relationships among them. This allows the modeller to understand and analyse new or existing organizational and system configurations even if the internal goals and beliefs of the individual agents are not known.

An example concerning a computer based training system (CBT) for volunteers of emergency services will be used to illustrate the SD model notation (see Figure 3.1 for the model). The modelling process begins with identifying the actors involved with the CBT system and their mutual dependency relationships (using the taxonomy of dependency relationships described above). The TrainingCo-ordinator agent
3.2. SD Model of the Computer Based Training System (CBT) for Volunteers

Figure 3.1: A Strategic Dependency model for computer based training system
depends on Volunteer agents to achieve its TrainingAttended goal. The class of Volunteer actors has a specialised sub-class of actors called SpeciallyTrainedVolunteers (these are volunteers who go through special training programs to acquire specialised skills). The TrainingCo-ordinator depends on SpeciallyTrainedVolunteers to achieve the goal SpeciallyDesignedTrainingAttended, modelled as a goal dependency. The TrainingCo-ordinator has two goal dependencies on the TrainingSystem: TrainingScheduled and OnlineTrainingConducted (i.e., the TrainingCo-ordinator agent relies on the TrainingSystem agent to schedule training sessions and to conduct online training). The TrainingSystem has a dependency on the TrainingCo-ordinator to provide TrainingContent, modelled as a resource dependency. The TrainingSystem has a dependency on Volunteers to achieve its TrainingAttended goal. The TrainingSystem has a dependency on Volunteers to provide Confirmation of their attendance, modelled as a resource dependency. Volunteers depend on the TrainingSystem to perform the ConductTraining task. Observe that we have chosen not to model this as a goal-dependency since the TrainingSystem cannot autonomously decide how the corresponding goal might be achieved but must work with the depender in a tightly-coupled fashion to perform the task. Volunteers have a further dependency on the TrainingSystem to TrainingScheduleReminder and TrainingInformation, modelled as resource-dependencies. Volunteers have a preference for the training system to satisfy the softgoal TrainingModulesEasy-ToUse.

3.3 SR Model of the Computer Based Training System (CBT) for Volunteers

In the i* framework, the SR model provides a more detailed level of modelling by looking “inside” actors to model internal intentional relationships. Intentional elements (goals, tasks, resources, and softgoals) appear in the SR model not only as external dependencies, but also as internal intentional elements linked by task-decomposition and means-ends relationships (Figure 3.2). The SR model in Figure 2 thus elaborates on the relationships between the TrainingCo-ordinator, TrainingSystem and Volunteer as represented in the SD model of Figure 3.1.

For example, the TrainingCo-ordinator, has an internal task to OrganiseTraining. This task can be performed by subtasks ScheduleTraining and GenerateTrainingContent (these are related to the parent task via task decomposition links). The GenerateTrainingContent task is further decomposed into subtasks SeekTrainingContent and
Figure 3.2: Strategic Rationale model for computer based training system

OrganiseTrainingContent. The softgoal TrainingContentEasyToUse is also related to the GenerateTrainingContent task via a task decomposition link. The intention is not to suggest that the softgoal plays the role of a sub-task but to relate the softgoal to the highest-level task for which the softgoal may be viewed as an optimisation objective. The softgoal thus serves to constrain design decisions on how the task might be decomposed. The subtask OrganiseTrainingContent is related to the TrainingContentEasyToUse softgoal via a “contributes to softgoal” link. In this instance, the contribution is positive, i.e., organising the training material contributes (positively) to achieving the broader goal of making the training material easy to use.

The TrainingSystem agent is identified with a high level rationale OrganiseTraining, modelled as a task. The task is further decomposed into the sub-tasks ImpartTraining, ObtainConfirmation and MaintainSchedule. The Volunteer agent is responsible for the task AcquireRelevantSkills, which is decomposed to the sub-tasks ParticipateInTraining and ProvideConfirmation.
3.4 Modelling Notation Driven Design of Requirements Capture Templates

Early-phase RE activities have traditionally been done informally [154], beginning with stakeholder interviews and discussions on the existing systems and rationales. Initial requirements are often ambiguous, incomplete, inconsistent, and usually expressed informally. We propose to add some structure to this informal consultation process via the use of Requirements Capture Templates (RCTs). In effect, these are forms that the modeller seeks to fill out in the course of a stakeholder consultation session and that must eventually be signed off by both the analyst/modeller and the stakeholder. The process of filling out these forms provides structure to stakeholder interview sessions. In addition, these forms have been designed to seek information specific to the needs of the underlying agent-oriented conceptual model that the analyst/modeller seeks to build. As we shall show below these templates have been designed in a manner that makes it easy to systematically transform them into SD and SR models. Stakeholders are thus able to provide focused input to the conceptual modelling task, while being shielded from the complexity of understanding and using the conceptual modelling language. In the following, we propose a methodology for requirements elicitation based on these requirements capture templates. The methodological guidelines that we offer are based on our experiences with early-phase requirements modelling of the emergency services organisation using the $i^*$ framework.

A key element of the process of agent-oriented conceptual modelling of a large organisation is the development of a hierarchically structured set of models. In our instance, we started with a highest-level SD model, which treated the emergency services organisation as a single agent that interacted with a variety of external entities and agents. A corresponding SR model was built which detailed the goals, tasks, resources and softgoals internal to the emergency services actor and their relationships to external actors and external dependencies. This SR model did not expand on the internal characteristics of the other actors in any great detail. At the next level of abstraction, we constructed an SD model of the emergency services organisation where each individual department was modelled as a distinct actor. The requirements capture templates (RCTs) became useful from this point onwards. Implicit in the RCTs presented here is the notion of an organisational model such as the one shown in Figure 3.3. Several concepts explicitly referred to in the RCTs, such as departments/units, functions and activities derived from such a model are extraneous to the $i^*$ notation. These notions
make the RCTs conceptually accessible to organisational stakeholders, yet at a lower level lead to notions directly supported by \( i^* \). The underlying organisational model helps make the elicitation process more systematic. Building such a model explicitly is helpful, but not essential (i.e., the underlying organisational model can remain implicit in the RCTs). The agents responsible for the activities as shown in Figure 3.3 may be internal or external to the organisation.

The first RCT (shown in Table 3.1 below) that we designed helped address the following questions:

- Why does the department exist?
3.4. Modelling Notation Driven Design of Requirements Capture Templates

- What are the department rationales?
- What are the main functions of the department?

The rationales and functions are revisited during the elicitation process by asking the ‘why-what-how’ questions till an agreement is reached on the requirements. The key role of this specific RCT is to identify the agents/actors that would form the basis of the SD and SR models to be constructed (involving department-level actors).

The next step involves elaboration of the high-level functions by identifying the various activities required to support each of the functions. The Function Elaboration Template (an instance of which is presented in Table 3.2) was designed (and used in the context of the emergency services organisation) to elicit information about specific functions within each department and activities supporting such functions. Note that one would fill out a form similar to the one shown in Table 3.2 for each activity supporting each function within each department. The specific slots in the template are self-explanatory, but the key point to note is that there is a relatively direct mapping from a collection of such completed templates to an SD model. The template identifies each of the actors to be represented in an SD model and provides information on the dependencies between them. The “relationship/dependency” column in the template provides, in effect, a name for the dependency. The “additional information/elaboration on the relationship” column can provide adequate pointers to appropriately classify the dependency (as a goal-dependency, task-dependency etc.). Information in this column can also provide an indication of how critical this dependency is (the SD model notation supports the representation of this information, although our example above does not illustrate it). Information on specialisation/generalisation relationships between actors can be obtained from a detailed analysis of the “source/target actor” columns. It is important that the modeller should only elicit the relationships/dependencies the source actor (stakeholder being interviewed) has on the target actor(s). Our empirical evidence suggests that the source actor may not have sufficient knowledge or be aware of the relationships/dependencies the target actor(s) has on the source actor. Hence, to complete the relationship/dependency between the remaining actors in the given activity we use the same requirements capture form and conduct a similar requirements gathering process with the remaining actors.

The next step involves identification of intentional relationships that are ‘internal’ for each actor for each activity described in the Function Elaboration Template. The Activity Elaboration Template (an instance of which is presented in Table 3.3) was designed to elicit information about specific activities within each actor. Note that one
Table 3.1: Example of Organisational Unit Template for Department Details

<table>
<thead>
<tr>
<th>Organisational Unit Template</th>
<th>Department Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Department Name</strong></td>
<td>Operations</td>
</tr>
<tr>
<td><strong>Name of the Department head</strong></td>
<td>Ms. Robyn M</td>
</tr>
<tr>
<td><strong>Designation of Department head</strong></td>
<td>Director</td>
</tr>
<tr>
<td><strong>Department Rationale</strong></td>
<td></td>
</tr>
<tr>
<td>• To provide all operations related function to external and internal stakeholders</td>
<td></td>
</tr>
<tr>
<td>• Use industry best practices in conducting the activities</td>
<td></td>
</tr>
<tr>
<td>• Keep up-to-date with industry standards</td>
<td></td>
</tr>
<tr>
<td>• Upgrade equipment with latest technologies</td>
<td></td>
</tr>
<tr>
<td><strong>High-Level Functions of the Department</strong></td>
<td></td>
</tr>
<tr>
<td>• Operations</td>
<td></td>
</tr>
<tr>
<td>• Planning</td>
<td></td>
</tr>
<tr>
<td>• Training</td>
<td></td>
</tr>
<tr>
<td><strong>Modeller Signature</strong></td>
<td><strong>Stakeholder Signature</strong></td>
</tr>
</tbody>
</table>
### Function Elaboration Template

#### Function Elaboration for the Department

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Name</strong></td>
<td><strong>Training</strong></td>
</tr>
<tr>
<td>(Use separate sheet for each function)</td>
<td></td>
</tr>
<tr>
<td><strong>Function Rationales</strong></td>
<td>To provide quality training programs</td>
</tr>
<tr>
<td>(Use separate sheet for each function)</td>
<td>To keep volunteers updated with latest tools and techniques</td>
</tr>
<tr>
<td></td>
<td>To provide fast and effective training programs</td>
</tr>
<tr>
<td></td>
<td>To conduct frequent training and certifications</td>
</tr>
<tr>
<td></td>
<td>Incorporate Industry best practices in training programs</td>
</tr>
<tr>
<td></td>
<td>Update training material</td>
</tr>
</tbody>
</table>

#### Activity Details for the Function

<table>
<thead>
<tr>
<th>Activity Name and Description</th>
<th>Training Program for Volunteers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Use separate sheet for each activity under the function)</td>
<td>(Provide training to volunteers on various emergency management topics)</td>
</tr>
</tbody>
</table>
### Activity Rationales
- Training forms a critical activity in emergency services
- There are specialised and starter programs
- Training is conducted every month
- All volunteers are supposed to undergo starter programs

### Responsible Actor(s) involved in the activity
(Unique list of Actor(s))

- Training Co-ordinator
- Training System
- Volunteer
  - Specially Trained Volunteer

### Relationship/dependencies between responsible actor(s) to achieve/satisfy the above activity

(Relationship is described as the dependency from source actor on to target actor, use separate row for each relationship and dependency)

<table>
<thead>
<tr>
<th>Source Actor</th>
<th>Relationship / Dependency</th>
<th>Target Actor</th>
<th>Additional information/elaboration on the relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Co-ordinator</td>
<td>Attend the Training</td>
<td>Volunteer</td>
<td>The Co-ordinator is finally responsible to ensure that the volunteers attend the program</td>
</tr>
<tr>
<td>Training Co-ordinator</td>
<td>To Schedule the training program</td>
<td>Training System</td>
<td>The Co-ordinator gives instructions to the system to schedule the training program</td>
</tr>
<tr>
<td>Training Co-ordinator</td>
<td>Carry out online training programs for volunteers</td>
<td>Training System</td>
<td>The Training Co-ordinator expects the training system to conduct the online training</td>
</tr>
<tr>
<td>Training Co-ordinator</td>
<td>Attend Specially Designed Training Programs</td>
<td>Specially Trained Volunteer</td>
<td>The Co-ordinator is finally responsible to ensure that the specially trained volunteers attend the training program</td>
</tr>
<tr>
<td>Training System</td>
<td>Provide with the training contents</td>
<td>Training Co-ordinator</td>
<td>To impart training the training system expects the Training Co-ordinator to provide content for the training</td>
</tr>
<tr>
<td>Training System</td>
<td>Attend Training</td>
<td>Volunteer</td>
<td>Training system depends on the Volunteer to attend the training</td>
</tr>
</tbody>
</table>
### 3.4. Modelling Notation Driven Design of Requirements Capture Templates

<table>
<thead>
<tr>
<th>Training System</th>
<th>Receive confirmation to attend the training program</th>
<th>Volunteer</th>
<th>The training system is dependent on the Volunteer to provide confirmation to attend the training program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volunteer</td>
<td>Conduct the training program</td>
<td>Training System</td>
<td>Volunteers expect the training system to conduct the online training program. The training modules should be easy to use</td>
</tr>
<tr>
<td>Volunteer</td>
<td>Provide Reminders to attend the training</td>
<td>Training System</td>
<td>Volunteers busy with their livelihoods, they would prefer to be reminded about the training programs</td>
</tr>
<tr>
<td>Volunteer</td>
<td>Provide information and plan on training programs</td>
<td>Training System</td>
<td>The training system must provide information to Volunteers on a query or inquire in regards to training programs or schedule.</td>
</tr>
</tbody>
</table>

would fill out a form similar to the one shown in Table 3.3 for each activity elaborating on the intentional characteristics that are ‘internal’ to the actors. This is done by identifying intentional descriptions of processes in terms of process elements and their rationale. The specific slots in the template are self-explanatory, but a key point to note is that there is a relatively direct mapping from a collection of such completed templates to an SR model. The template identifies each of the actors with their internal characteristics that provide an understanding of the process elements that could be classified as a goal, task, resources, or softgoal. The “internal task/means” column in the template provides, in effect, names for the internal characteristics. The “additional information on tasks/means” column can provide adequate pointers to appropriately classify the internal characteristic (as a goal, task, resource, and/or softgoal) and infer how each high-level task internal to an actor might be decomposed (either into sub-tasks or into means to achieve the task). Information in this column can also provide an indication of how a particular internal characteristic can provide a positive or negative contribution to the other internal characteristics (such as a sub-task supporting a high-level task or a task positively contributing to a softgoal associated with a higher-level task). It is important that the modeller should only elicit the intentional
Table 3.3: Example of Activity Elaboration Template for Internal Intentional Characteristics of individual actor(s) to achieve the activity

<table>
<thead>
<tr>
<th>Activity Elaboration Template</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal Intentional Characteristics of individual actor(s) to achieve the activity</strong></td>
</tr>
<tr>
<td>(Use multiple rows to describe multiple internal tasks for each actor)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Name</td>
<td>Training</td>
</tr>
<tr>
<td>Activity Name</td>
<td>Computer-based Training Program for Volunteers</td>
</tr>
</tbody>
</table>

**Responsible Actor(s) involved in the activity**
(Unique list of Actor(s))
- Training Co-ordinator
- Training System
- Volunteer
  - Specially Trained Volunteer

<table>
<thead>
<tr>
<th>Actor</th>
<th>Internal Task / Means to achieve the activity by individual Actor</th>
<th>Additional information on task or means to achieve the activity or Actor Rationales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Co-ordinator</td>
<td>• Organise Training Programs</td>
<td>The co-ordinator is responsible for organising the entire training programs.</td>
</tr>
</tbody>
</table>
3.4. Modelling Notation Driven Design of Requirements Capture Templates

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>Generate information regarding training and its content</td>
<td>To organise training programs would result in generating training material and content for the various training programs that must be easy to use by volunteers or users of the content</td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>Acquire and seek information regarding training program and its content</td>
<td>To generate quality training content would result in seek for content from various sources for the training programs</td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>Organise the collected Training Content</td>
<td>To generate and provide an easy to use training content would lead to organising the content in an acceptable fashion for use.</td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>Schedule the training program</td>
<td>The co-ordinator is responsible for scheduling and conducting the training program with the help of a computer based tool.</td>
<td></td>
</tr>
</tbody>
</table>

| Modeller Signature | Stakeholder Signature |
characteristics that are ‘internal’ to the actor (stakeholder being interviewed). Our empirical evidence suggests that the source actor may not have sufficient knowledge of the intentional characteristics that are ‘internal’ to other actors in the given activity. Hence, to complete the intentional characteristics that are ‘internal’ for the remaining actors/agents in the given activity it is proposed to use the same requirements capture form and conduct a similar requirements gathering process with the remaining actors.

We have argued that the templates presented here can ease the requirements elicitation process. However, these templates serve other useful functions as well. They can provide a structured repository and record of stakeholder interviews that can be revisited when requirements must be re-negotiated or revised (for instance, when changes are made to models, or when inconsistencies are detected). The detailed rationale recorded in these templates can also be of value in business process re-engineering. To anticipate and support future business process re-engineering efforts in the context of the emergency services agency, we are also detailing alternative solution scenarios by completing additional RCTs that answer “how-else” questions (while the primary RCTs represent the “as is” scenarios). Examples of the actual RCTs used in the elicitation of requirements of emergency services agency are provided in Appendix A.

3.5 An Ontology Driven Approach to Requirements Elicitation

An ontology is commonly understood to be a description (like a formal specification of a program) of concepts and relationships that can exist for an agent or a community of agents. The notion of ontology, as used in computing, refers to a common vocabulary (with a concomitant set of rules) that is used for building and reasoning about systems. [81] presents a good survey of ontology-based approaches to information systems development. A variety of domain ontologies have been developed and several are publicly available [51, 52, 83]. The study, development and deployment of ontologies have received considerable recent attention as a result of the semantic web initiative [127]. Several ontology languages (which serve the dual role of ontology markup languages) have been developed, including the Ontology Inference Layer (OIL) [105] and the DARPA Agent Markup Language (DAML) [30]. The semantic web initiative has led to a large number of web-based ontologies being developed, through what may be viewed as a large distributed collaborative knowledge engineering exercise. It is therefore not unreasonable to assume that analysts and application developers would have
access to reusable enterprise ontologies as well as reusable function/activity-specific ontologies. A simple approach to conceptualising an ontology is to view it as a concept vocabulary coupled with a set of rules. The rules may be structural rules that may, for instance, organise concepts in a class hierarchy, or they may be semantic constraints or business rules (an example is a rule in a banking application that requires interest rates for loan accounts to be always higher than those for savings accounts).

We propose to exploit the availability of such reusable ontologies in our approach to early-phase requirements engineering via agent-oriented conceptual modelling. Our key premise is that a pre-existing knowledge-base (or even a concept vocabulary) can significantly ease the early-phase requirements modelling task (by providing some modicum of guidance to a modeller who might be venturing into the task with no prior knowledge or understanding of the application domain). A pre-existing domain ontology can thus help provide focus to a modeller’s early interactions with stakeholders. However, our proposal here is to formalise the process by which ontology-driven elicitation might take place. This is then generalised into a full ontology life-cycle in the requirements elicitation context.

Recall that we are interested in ontologies of two distinct kinds: enterprise ontologies and function/task-specific ontologies. Enterprise ontologies can provide guidance in
identifying actors while constructing high-level SD and SR models, by making available certain default organisational structures. These can also provide a vocabulary for more refined (lower-level) SD and SR models. Function/task-specific ontologies (often included within enterprise ontologies) provide detailed concept vocabularies for specific tasks, which can serve as elicitation triggers.

Ontologies can also provide a benchmark for completeness that serve to drive the elicitation process. Informally, a conceptual model is deemed to be complete with respect to an ontology if it makes reference to every concept in the concept vocabulary of the ontology. This is in many ways analogous to the notion of completeness of formal theories. A theory is considered complete with respect to a language if it commits to the truth or falsity of every proposition in the language. It is not difficult to conceive of an elicitation methodology that uses this notion of completeness of a conceptual model relative to an ontology to generate elicitation triggers. In effect, every instance of incompleteness, i.e., every concept in the concept vocabulary that is not referred to by the conceptual model, serves as a trigger for further questions/probes from the modeller.

Ontologies can also support consistency testing of conceptual models. A conceptual model would be deemed inconsistent relative to an ontology if it violated any of the rules associated with the ontology. These could be violations of the structural rules (for instance if a subclass-supersclass relationship is reversed in a model) or violations of semantic constraints (for instance, an activity that involves an actor making his/her appointments schedule publicly available may violate security constraints). Each instance of inconsistency can serve as an elicitation trigger, obliging the modeller to seek out additional information in the process of resolving the inconsistency (usually by appropriately modifying the conceptual model).

Much of our discussion above assumes that appropriately constructed domain ontology is made available to the modeller at the start of the elicitation phase. This can be an unrealistic assumption since pre-existing ontologies, where available, may turn out to be inadequate. Key concepts from the domain may not be included in the concept vocabulary, while key relationships may not be represented in the rule-set. The challenge, then, is to devise early-phase requirements modelling methodologies that maintain and update ontologies. These same methodologies might also be used to build (if necessary, from scratch) appropriate domain ontologies. Figure 3.4 provides the outlines of such a methodology. The key points to note are as follows. Arrow from one entity to another indicates the ability of the former to contribute to the process of
building (and updating) the latter. Thus, ontologies can help build conceptual models, as discussed above, but the reverse is also true. The process of building a conceptual model can reveal gaps in an ontology - most obviously in its concept vocabulary, but conceivably also in its set of rules. An ontology can help design a requirements capture template, but completed templates can similarly suggest gaps in ontologies. A requirements capture template can serve as a basis for conceptual model building (as described in the previous section, but a conceptual model can suggest alternative ways of structuring requirements capture templates.

3.6 Summary

This chapter presented a robust early-phase requirements elicitation methodology. We have argued for the use of elicitation templates that reflect the information requirements of the underlying modelling notation to ease the elicitation process. We have described how an organisational model that assists modellers and stakeholders during the requirements gathering process (interview process). We have also argued for the use of enterprise ontologies to drive the elicitation process. These proposals can be generalised to other conceptual modelling contexts (using distinct modelling notations to the one being used in our proposal).

In the next chapter, we shall give a proposal about the combined use of agent-oriented conceptual modelling with formal methods. This chapter explores how these two complementary approaches might be used in a synergistic fashion for requirements engineering.
Chapter 4

Combining Agent-Oriented Conceptual Modelling with Formal Methods

This chapter describes how $i^*$ modelling framework and Z notation can function in a complementary and synergistic fashion for Requirements Engineering. The proposed methodology is exemplified by a Flood Rescue Management case study (this is part of a collaborative project to build a comprehensive enterprise model for an emergency services agency). This chapter may be viewed as a first step in defining a complete methodology for supporting the co-evolution of $i^*$ models and Z specifications. We then introduce the notion of refinement with additional information from the existing SD and SR models (i.e. information not included in an $i^*$ model, but obtained via further analysis e.g., temporal sequencing of dependencies, fulfillment conditions for dependencies etc).

4.1 Introduction

The $i^*$ framework is an agent- and goal-oriented modelling language that has been specifically designed for early-phase requirements engineering and that is well-suited for answering questions such as these. The $i^*$ framework is particularly useful for:

- making explicit (and in the process gaining) a deeper understanding of the organisational relationships between various actors in the target environment
- understanding of the rationale behind the existing practices and structures and
- representing, at an intentional level, the internals of actors populating the target system, and relating these explicitly to organisational objectives and inter-actor relationships.
4.1. Introduction

Several proposals exist for integrating $i^*$ modelling with late-phase requirements analysis and the downstream stages of the software life-cycle. The Tropos methodology [24] explores how $i^*$ models might be refined to form the basis for late-phase requirements specifications, and subsequently architecture specifications. As pointed out earlier that, $i^*$ framework helps in answering questions like: what are the main goals of the system, how stakeholders depend on each other, and what alternatives exist [157]. The emphasis here is on understanding the “whys” that underlie system requirements, rather than on the precise and detailed specification of “what” the system should do [159]. On the other hand, the late-phase of requirements engineering focuses on completeness, consistency and automated verification of requirements [57, 18]. The $i^*$ notation alone is not adequate for representing the level of detail necessary for late-phase requirements specifications. To address this problem, a custom-designed formal languages called Formal Tropos [55] have been developed. Alternative approaches have been proposed to define methodologies for transforming $i^*$ models into agent programs in formal agent programming languages such as ConGolog [145], AgentSpeak(L) [125] and 3APL [61].

While Z can be used for early-phase requirements modelling, the necessary level of formalisation, precision and detail, the lack of a diagrammatic notation to support the visualisation of requirements and the inability to represent the intentional elements all suggest that an alternative notation such as $i^*$ might be better suited for this phase. Our aim is to support the modelling of organisational contexts, intentions and rationale in $i^*$, while traditional specifications of functionality and design proceeds in the formal notation. In this chapter, we focus on Z as a prototypical representative of a formal notation, but observe that many of the lessons generalise to other formal methods. More generally, this research suggests how diagrammatic notations for modelling early-phase requirements, organisation contexts and rationale can be used in a complementary manner with more traditional specification notations.

Our proposal for a synergistic combination of $i^*$ and Z offers several advantages:

- $i^*$ and Z can be viewed as a pair of complementary representation languages that can be jointly brought to bear on the requirements engineering exercise. The $i^*$ notation permits us to make explicit the intentional aspects of the requirements specification, including an understanding of the organisational context of the proposed system, the alternatives that may be considered in making design decisions as well as the rationale behind these decisions (these latter features support process reengineering). The Z notation permits us to specify late-phase
requirements with a degree of precision and formality that \( i^* \) does not.

- The \( i^* \) notation allows us to represent and reason with softgoals (representations of non-functional requirements or objectives).

- We propose a mapping from \( i^* \) models into Z schemas that does not result in any information loss to the original \( i^* \) model (this is distinct from proposals such as the one involving mapping \( i^* \) models to ConGolog agent programs [145], where aspects of the \( i^* \) model are ignored in the translation).

- The mapping of \( i^* \) models to Z schemas enables the refinement of these schemas with additional information, such as invariant properties, fulfillment conditions etc. (note that these cannot be represented in the original \( i^* \) model).

- Current approaches to the use of formal methods in conjunction with \( i^* \) models are unduly complicated. Formal Tropos [55], for instance, is an intermediate language in which \( i^* \) models must be defined before an eventual translation into a state machine model on which model checkers can be deployed to verify systems properties (the process also assumes a significant amount of refinement of the original model with additional information). Existing tool support for Z, on the other hand, allows analysis of specifications without any of this additional effort.

- Bissener has studied the combined use of the \( i^* \) and Albert-II frameworks for requirements engineering in [9]. The process models do not specifically show how the whole system proceeds step by step (as compared to the approach presented in this thesis). Agents in Albert-II do not have intentional characteristics and they do not have goals. The work only focuses on specification, and is not concerned with the examination of alternatives for meeting goals. While some tool support exists for Albert-II [46], use of Albert-II is not widespread. Tool support is not mature enough as Z notation.

- CIMOSA (Computer Integrated Manufacturing Open System Architecture) [89, 143] represents an enterprise modelling architecture. CIMOSA recommends performing an information analysis based on the object-oriented paradigm. In contrast to the approach presented in this thesis, it does not support the modelling of organisational contexts, intentions and rationale. Combined use of \( i^* \) and Z offers greater expressive power than CIMOSA. For example in CIMOSA, we cannot describe actors/agents, dependencies, intentional elements while we can
express these in the methodology presented in this thesis. At the same time, Z has found acceptance in industry [13].

4.2 A Formal basis for Co-evolution of Models

A key concern of this research is the definition of methodologies that permit models in distinct notations (ideally with complementary representational capabilities) to co-evolve. Informally, models in distinct notations are said to co-evolve if the following are true:

- These models are independently maintained and updated, possibly by distinct sets of stakeholders.
- At any given point in time, these models satisfy a set of formal or informal inter-model consistency constraints. In other words, the models in distinct notations must not represent contradictory descriptions of the same reality.

The key to formalising this notion is to understand inter-model consistency constraints. These are, in general, hard to obtain and must be hand-crafted for every pair of notations of interest. A more achievable approach involves, for a pair of models in distinct notations:

- Mapping each model into a model in the other notation.
- Using the consistency rules or semantics intrinsic to each notation to determine if the models are consistent. These consistency rules or semantics are usually easier to come by and even when they are not formally specified, consistency is relatively easy to manually verify.

Formally, this can be achieved by defining a mapping function in the following manner. Let \( N_1 \) and \( N_2 \) be two distinct modelling notations. Let \( f_{N_1,N_2} : M_{N_1} \rightarrow M_{N_2} \) where \( M_{N_1} \) and \( M_{N_2} \) are the sets of all possible models expressible in \( N_1 \) and \( N_2 \) respectively, be a function that maps a model in \( N_1 \) to a model in \( N_2 \). Ideally, such a function must generate an \( N_2 \) model that expresses as much of the input model (in \( N_1 \)) as can be expressed in \( N_2 \). \( f_{N_1,N_2} \) is similarly defined. Co-evolution of models in \( N_1 \) and \( N_2 \) can then be defined as follows:

If \( M_1 \) and \( M_2 \) are the current models in \( N_1 \) and \( N_2 \) respectively, then it must be the case that \( f_{N_1,N_2}(M_1) \) is consistent with \( M_2 \) and \( f_{N_2,N_1}(M_2) \) is consistent with \( M_1 \).
4.3 Mapping \(i^*\) into \(Z\)

4.3.1 Mapping a general SD model into \(Z\)

Key steps in the proposed methodology are listed in Table 4.1. The sets of all actor names, \(all\_actors\), and dependency names, \(all\_depend\), are defined as power sets of the set \(NAME\). The notion of \(STATE\) is implicit in \(i^*\), but requires explication in \(Z\) specifications.

\[
\begin{align*}
\text{[NAME]} \\
| \text{all\_actors, all\_depend} : \mathbb{P}_1 NAME
\end{align*}
\]

It is necessary to mention that names of internal intentional elements of a SR model are also members of the given set \(NAME\) but do not belong to subset \(all\_depend\). Formalisation of these internal intentional elements is considered later in the chapter. Both SD and SR models provide a description of the intentional relationships among actors of a process and do not directly address the dynamics of this process. But exactly the dynamics are the most important for process or system specification. To reflect it, we use the fact that all dependencies in SD and internal intentional elements in SR are realised dynamically: a goal is achieved, task is performed or resource becomes available. We consider different states of the dependencies (elements) before and after realisation using the following free type definition:


4.3. Mapping $i^*$ into $Z$

Table 4.1: Key steps in the proposed methodology of mapping $i^*$ into $Z$

<table>
<thead>
<tr>
<th>Key steps in the proposed methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Map a general SD model into $Z$</td>
</tr>
<tr>
<td>(i). Identify sets of all actor names, all actors, and dependency names, all depend, define them as power sets of the set NAME.</td>
</tr>
<tr>
<td>(ii). Define states of dependencies (elements) before and after their realisation (free type definition in $Z$).</td>
</tr>
<tr>
<td>(iii). Create SD schema—which describes state of the whole SD model as the set of states of all its dependencies.</td>
</tr>
<tr>
<td>(iv). Define each SD dependency and SR element with types and degrees (using free type definition in $Z$).</td>
</tr>
<tr>
<td>(v). Formalise all the dependencies in the SD diagram using following steps:</td>
</tr>
<tr>
<td>• Create $\Phi$Depend schema which describes a common pattern of SD dependencies and SR elements.</td>
</tr>
<tr>
<td>• Create $S$Dependency schema which describes a general structure of all the SD dependencies and includes $\Phi$Depend schema as one of the component part.</td>
</tr>
<tr>
<td>• Create detailed schema for every SD dependency using $S$Dependency schema as a basis.</td>
</tr>
<tr>
<td>2. Map a general SR model into $Z$.</td>
</tr>
<tr>
<td>(i). Create Actor schema which describes a general structure of all the actors in SR diagrams.</td>
</tr>
<tr>
<td>(ii). Create $A$Element schema which describes a general structure of all the SR internal intentional elements and includes $\Phi$Depend schema as one of the component part.</td>
</tr>
<tr>
<td>(iii). Create a detailed schema for every actor in the specific SR model using Actor schema as a basis.</td>
</tr>
<tr>
<td>(iv). Create a detailed schema for every internal intentional element of every actor using $A$Element schema as a basis.</td>
</tr>
</tbody>
</table>
### 4.3. Mapping \(i^*\) into \(Z\)

\[
STATE ::= \text{inapplicable} \mid \text{unresolved} \mid \text{fulfilled} \\
\quad \mid \text{violated} \mid \text{satisficed} \mid \text{denied} \mid \text{undetermined}
\]

State inapplicable is held before the creation of a new instance of a dependency (element). State unresolved conforms to a dependency (element) after the creation but before realisation and all other states are conforming to a dependency (element) after realisation. The dependency (element) is in state fulfilled if realisation is successful and in state violated if realisation is unsuccessful. With the idea of keeping uniform terminology with other researchers (e.g., [28]) in the area, for softgoals we use two states satisficed and denied. The last state undetermined can also be used only for a softgoal. Softgoals are often identified with quality criteria and sometimes it is impossible to conclude immediately after realisation whether a quality criterion is satisfied. It means that it may not be clear whether the realisation had been successful or not. In this case we consider the softgoal is in the undetermined state.

The state of a whole SD model is the set of states of all its dependencies for this SD model that is reflected in \(SD\) schema\(^1\):

\[
\begin{align*}
SD & \\
SD\_state & : NAME \rightarrow STATE \\
\text{dom } SD\_state & = \text{all\_depend}
\end{align*}
\]

Thus, the realisation of a dependency changes its state and at the same time changes a state of the whole SD model.

Each SD dependency or SR element has its own specific features and differs first in types and degrees.

\[
\begin{align*}
TYPE & ::= \text{goal} \mid \text{softgoal} \mid \text{task} \mid \text{resource} \mid \text{ISA} \\
\text{DEGREE} & ::= \text{open} \mid \text{committed} \mid \text{critical}
\end{align*}
\]

In contrast to other values, the \(ISA\) type does not represent a dependency. It means that one actor can be considered as a special instance of other actor. Since, \(ISA\) is a relationship between two actors it is convenient for us to consider them together as a different values of \(TYPE\). All other values of free type definitions \(TYPE\) and \(DEGREE\) are standard for the \(i^*\) framework.

All the dependencies in SD (as well as every element in SR model) are described by its own schema. A general structure of SD dependencies (external between actors) varies from a general structure of SR elements (inside actors) but at the same time they

\(^1\)All schemas in this thesis are checked using the ZTC type-checker package [74].
4.3. Mapping $i^*$ into Z

have some common patterns. That is why we use the following steps of formalisation, creating successively:

- $\Phi_{\text{Depend}}$ schema which describes a common pattern of SD dependencies and SR elements; (the $\Phi_{\text{Depend}}$ in the schema name is used to flag a partial specification [132]).

- $\text{SDependency}$ schema which describes a general structure of all the SD dependencies and includes $\Phi_{\text{Depend}}$ schema as one of the component part;

- A detailed schema for every SD dependency using $\text{SDependency}$ schema as a basis.

Common patterns for SD dependencies and SR elements are represented in $\Phi_{\text{Depend}}$ schema. Here, $\Phi$ is a part of the schema name, not an operator. It is just a naming convention used to indicate a partial (incomplete) specification [13].

$$
\Phi_{\text{Depend}}
$$

<table>
<thead>
<tr>
<th>dependum : NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>type : TYPE</td>
</tr>
<tr>
<td>degree : DEGREE</td>
</tr>
<tr>
<td>result! : STATE</td>
</tr>
<tr>
<td>result! ≠ unresolved</td>
</tr>
<tr>
<td>result! = satisfied $\lor$ result! = denied $\lor$ result! = undetermined $\Rightarrow$ type = softgoal</td>
</tr>
</tbody>
</table>

Except for the above-mentioned type and degree, specific features of every dependency are its name ($\text{dependum}$) and resulting state, which is represented by the output variable $\text{result!}$.

- The first line of the predicate part of $\Phi_{\text{Depend}}$ describes the fact that the resulting state cannot be $\text{unresolved}$.

- The second line of the predicate part of $\Phi_{\text{Depend}}$ reflects that the resulting state can take the $\text{satisfied}$, $\text{denied}$ or $\text{undetermined}$ value only for softgoals.

The following $\text{SDependency}$ schema is a result of one-to-one mapping of the general structure of a SD dependency into the Z notation. This schema is an operation schema and changes the state of the SD model ($\Delta SD$). $\text{SDependency}$ schema includes the components $\Phi_{\text{Depend}}$ schema as well as names of actors ($\text{depender}$ and $\text{dependee}$) which
are linked by the dependency. This schema also includes the names of the internal intentional elements \((\text{dependee}_\text{internal}_\text{element} \text{ and } \text{dependee}_\text{internal}_\text{element})\) linked to the dependency. The sets \(\text{actor}_\text{element}_\text{dependee} \text{ and } \text{actor}_\text{element}_\text{dependee} \) are the names of all the internal intentional elements present in the \text{dependee} and \text{dependee} respectively. While this schema represents a general structure, its name, type, degree and names of actors are not specified. It could be done later on during the consideration of an \(i^*\) model for a specific example.

\[
\begin{align*}
\text{SDependency} \\
\Delta SD \\
\Phi \text{Depend} \\
\text{dependee}, \text{dependee} : \text{NAME} \\
\text{dependee}_\text{internal}_\text{element}, \\
\quad \text{dependee}_\text{internal}_\text{element} : \text{NAME} \\
\text{actor}_\text{element}_\text{dependee} : \mathbb{P}_1 \text{NAME} \\
\text{actor}_\text{element}_\text{dependee} : \mathbb{P}_1 \text{NAME} \\
\text{dependum} \in \text{all}_\text{depend} \\
\text{dependee} \in \text{all}_\text{actors} \\
\text{dependee} \in \text{all}_\text{actors} \\
\text{dependee}_\text{internal}_\text{element} \in \text{actor}_\text{element}_\text{dependee} \\
\text{dependee}_\text{internal}_\text{element} \in \text{actor}_\text{element}_\text{dependee} \\
\text{SD}_\text{state'} = \text{SD}_\text{state} \oplus \{\text{dependum} \mapsto \text{result!}\}
\end{align*}
\]

The most significant information is contained in the last line of the predicate part of this schema, which describes how the realisation of the dependency changes the state of the SD model. Using the \textit{override} operator \(\oplus\) shows that the value of the SD model’s state function \(\text{SD}_\text{state'}\) after the dependency realisation differs from its value \(\text{SD}_\text{state}\) before the realisation only in the part of the considered dependency and coincides for all other dependencies.

### 4.3.2 Mapping a general SR model into Z

Our approach of mapping a SR model into the Z notation is similar to the approach for SD diagrams which were considered in the last section. The mapping consists in consecutively creating:

- \textit{Actor} schema which describes a general structure of all the actors in SR diagrams
- \textit{AEElement} schema which describes a general structure of all the SR internal intentional elements and includes \(\Phi \text{Depend}\) schema as one of the component part
4.3. Mapping $i^*$ into $Z$

- A detailed schema for every actor in the specific SR model using $Actor$ schema as a basis

- A detailed schema for every internal intentional element of every actor using $AElement$ schema as a basis.

The following schema describes a general structure of all the actors. The state of an actor is given by the set of states of all its internal (SR) intentional elements (i.e., goals, tasks etc). An actor is characterised by its name $actor\_name$, set $actor\_element$, names of all the internal intentional elements, and state function $actor\_state$.

<table>
<thead>
<tr>
<th>$Actor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$actor_name : NAME$</td>
</tr>
<tr>
<td>$actor_element : \mathbb{P} \ \NAME$</td>
</tr>
<tr>
<td>$actor_state : NAME \to STATE$</td>
</tr>
</tbody>
</table>

$$actor\_name \in all\_actors$$

$$\text{dom } actor\_state = actor\_element$$

The $actor\_state$ function is similar to the SD model’s state function $SD\_state'$ and represents a collection of states of all internal intentional elements of the actor.

For formalising a general structure of all SR elements, we need to introduce a new free type, which describes possible types of links between the elements.

$$LINK\_TYPE ::= NA \mid task\_decomp \mid means\_ends \mid contrib$$

Type $NA$ (Non-Applicable) is used for elements which have no means for attaining them and have no components. Type $task\_decomp$ represents task decomposition links. Types $means\_ends$ and $contrib$ describe means-ends links. Type $means\_ends$ is used for Goal-Task (GT) Link, Resource-Task (RT) Link, Softgoal-Task (ST) Link and Softgoal-Softgoal (SS) Link respectively. Type $contrib$ represents special kinds of means-ends links for softgoal (ST and SS links).

For convenience, we allocate all conditions connected with links into a separate schema $Link$. This schema includes:

- Names of internal (inside the actor) elements $int\_components$ which are linked with the considered element

- Names of external (from SD model) dependencies $ext\_components$ which are linked with the considered element
• Type of the link

• Names of elements which give positive ($contrib_p$) and negative ($contrib_m$) contribution to the softgoals.

\[
\begin{align*}
Link & \\
\Phi \text{Depend} & \\
\text{int}_\text{components}, \text{ext}_\text{components} : \mathbb{P} \text{NAME} & \\
\text{contrib}_p, \text{contrib}_n : \mathbb{P} \text{NAME} & \\
\text{link} : \text{LINK}\_\text{TYPE} & \\
\text{link} = \text{task}_\text{decomp} \Rightarrow \text{type} = \text{task} & \\
\text{link} = \text{contrib} \Rightarrow \text{type} = \text{softgoal} & \\
\text{contrib}_p \cup \text{contrib}_n \neq \emptyset \Rightarrow \text{link} = \text{contrib} \land & \\
(\text{contrib}_p, \text{contrib}_n) \text{ partitions int}_\text{components} & \\
\text{ext}_\text{components} \neq \emptyset \Rightarrow \text{type} = \text{task} & \\
\text{link} = \text{NA} \Leftrightarrow \text{int}_\text{components} = \emptyset & 
\end{align*}
\]

The predicate part describes the following constraints between types of links and types of elements:

• Task decomposition links are used only for tasks

• Positive or negative contribution is possible only for softgoals

• Tasks are used in connection with external components.

The following schema describes a general structure of all the SR internal elements. This operational schema changes the state of the general model of an actor ($\Delta\text{Actor}$). Similar to $S\text{Dependency}$ schema, $A\text{Element}$ schema includes $\Phi\text{Depend}$ schema as component. Inclusion of $Link$ schema brings all the information concerning links between the elements.

\[
\begin{align*}
A\text{Element} & \\
\Delta\text{Actor} & \\
Link & \\
\text{dependum} \in \text{actor}_\text{element} & \\
\text{int}_\text{components} \subseteq \text{actor}_\text{element} & \\
\text{ext}_\text{components} \subseteq \text{all}_\text{depend} & \\
\text{actor}_\text{name}' = \text{actor}_\text{name} & \\
\text{actor}_\text{element}' = \text{actor}_\text{element} & \\
\text{actor}_\text{state}' = \text{actor}_\text{state} \oplus \{\text{dependum} \mapsto \text{result}\!\!\!\!\!\}
\end{align*}
\]
The predicate part of $AEElement$ schema formalises the changes of $Actor$ schema under the realisation of the internal intentional element. Only one component of $Actor$ schema namely the actor’s state function $actor\_state'$ is changed. Similar to the SD model’s state function $SD\_state$, the difference between the values of $actor\_state$ before and after the element realisation exists only in the state of the considered element.

### 4.4 Case Study: Flood Rescue Management

This chapter presents a case study based on a collaborative project to build a comprehensive enterprise model for an emergency services agency (ESA). The case study concentrates on a key function of the ESA: managing flood rescue and evacuation operations.

The ESA is responsible for managing diverse emergency situations. The case study deals with an event, from an ESA perspective, is a flood response operation. The timing of the emergency response is critical in these scenarios. The ESA is the agency designated by the government to deal with these kinds of situations since it has the expertise and appropriate resources to deal with the threat.

During this emergency situation, an Emergency Coordination Centre is formed and the Coordinator (ECCC) heads it. The first action taken by the coordinator is to activate the emergency plan partially or fully depending on the situation. The main function of the coordinator is to bring together elements of the organisation together to ensure effective emergency management response and is primarily concerned with the systematic planning and application of resources (manpower and equipment). ECCC is responsible for the acquisition of additional resources requested by different Field Control Centre Coordinators (FCCC). Other responsibilities of ECCC include, to collect and assess field information so that rescue and evacuation operation can be coordinated in an efficient manner. Analyse weather forecast supplied by weather bureau and forward the analysed forecast to the concerned FCCC with necessary comments/observations.

The Field Control Centre is a facility where the FCCC is located, usually near the scene of an emergency, to facilitate control and management of the emergency. The FCCC is primarily responsible for managing the rescue and evacuation operation in the flood affected area. The FCCC is also responsible for publicising evacuation routes for the community, managing volunteers and available resources at his disposal in most optimal way.
The other people involved in the case study are Call Taking Supervisor/System, Volunteers/Emergency Workers, Community and Weather Bureau. Call Taking Supervisor/System is responsible for managing/handling calls from the affected people, classifying/prioritising them and forwarding calls to concerned authorities for further action. Volunteers/Emergency Workers are very important actors in the whole emergency situation. They are trained in all aspects of rescue operation. They are proficient in general rescue, providing first aid, operating communication equipment, map reading and navigation, flood rescue boat operations, giving storm safety advice, provision of essentials to people cut off by flood waters etc. Community actors in our case study are people who are affected by the flood. They are the people who are living in the flood-prone area. They are concerned about many issues and would like to know the answers of following questions:

- How deep could the water get in and around the property?
- Whether I might need to evacuate or will I be cut off by flooding in the area?
- Which are the safest evacuation routes?

The volunteers and the ESA provide the answers to these questions. Weather Bureau is responsible for providing weather forecast data that is crucial for the efficient planning of emergency operation. The other people involved in rescue and evacuation mission are dependent on weather bureau to provide forecast data at regular interval for the duration of emergency.

The SD model provides an external characterisation of an actor/agent in terms of two sets of dependencies: incoming dependencies (the agent acting as dependee) and outgoing dependencies (the agent acting as depender) [156]. In SD model this is represented by showing the left half-arrow (pointing from right to left) to denote incoming dependency and the right half-arrow to denote outgoing dependency. The case study shown in Figure 4.1 is used to illustrate the SD model [157] of managing flood rescue and evacuation operations by ESA. The modelling process begins with the identification of actors/agents involved in the flood rescue and evacuation operation and identifying mutual relationships between them. The EmergencyCoordinationCentreCoordinator (ECCC) agent depends on the FieldControlCentreCoordinator (FCCC) agents to accomplish its goal RescuePeopleAtRisk. Similarly the FCCC agent depends on the ECCC agent to achieve CoordinationSupport goal. The ECCC depends on the CallTakingSupervisor/System to provide InformationAboutPeopleAtRisk, modelled as
a goal dependency. Similarly, ECCC agent depends on WeatherBureau and Volunteers/EmergencyWorkers agents to achieve the goals WeatherForecast/Warnings and Evacuation and RescueMission respectively. The ECCC has a dependency on the WeatherBureau to provide WeatherData, modelled as a resource dependency. Similarly, ECCC has a dependency on the Volunteers/EmergencyWorkers and FCCC agents to provide FieldInformation and AcknowledgmentOfEmergencyNotification, LocalInformationUpdate respectively, modelled as resource dependencies. The remaining dependencies may well be explained on the similar lines.

An SR model provides a more detailed level of modelling by looking “inside” actors to model internal intentional elements such as goals, tasks, resources, and softgoals which appear in an SR model not only as external dependencies, but also as internal intentional elements linked by task-decomposition and means-ends relationships. The case study shown in Figure 4.2 is used to illustrate the SR model [157] of managing flood rescue and evacuation operations by ESA. Also shown are SR models to show the internal intentional elements of the actors/agents namely, CallTakingSupervisor/System (Figure 4.3), Volunteers/EmergencyWorkers (Figure 4.4), EmergencyCoordinationCentreCoordinator (Figure 4.5), FieldControlCentreCoordinator (Figure 4.6) and Community (Figure 4.7) respectively.

For example, Volunteers/EmergencyWorkers has an internal task to RescuePeople. This task can be performed by subtasks PrepareForRescue, MapReading&Navigation, OperateRescueBoats, CommunicationEquipmentOperation, SupplyEssentials, Rescue/EvacuatePeopleatRisk and the goal ReportSituation (modelling this as a goal instead of a task suggests that several alternative ways of achieving the goal exist and no commitment has been made to any single one of these). Each of these tasks and goals are related to the parent task via task decomposition links. The Rescue/EvacuatePeopleatRisk task is further decomposed into subtasks ProvideFirstAid and ConductEmergencyDrills. The softgoal Fast&Efficiently is also related to the RescuePeople task via a task decomposition link. When a softgoal is a component in task decomposition, it serves as a quality goal for that task. Assessment of the local flood situation by Volunteers/EmergencyWorkers in the flood-affected region is crucial for the ECCC and FCCC agents to further plan their resources and strategies in an optimal way. This is represented as subgoal ReportSituation in the SR model of Volunteers/EmergencyWorkers. This subgoal can be achieved by using any one of the shown subtasks, Radio/MobilePhone or SatellitePhone (this is represented by a means-ends
Figure 4.1: The Strategic Dependency Model of the Flood Rescue Management case study
4.4. Case Study: Flood Rescue Management

Figure 4.2: The Strategic Rationale Model of the Flood Rescue Management case study
Figure 4.3: The Strategic Rationale Model of the Actor Call Taking Supervisor/System
Figure 4.4: The Strategic Rationale Model of the Actor Volunteers/Emergency Worker
Figure 4.5: The Strategic Rationale Model of the Actor Emergency Coordination Centre Coordinator
4.4. Case Study: Flood Rescue Management

Figure 4.6: The Strategic Rationale Model of the Actor Field Control Centre Coordinator

Figure 4.6: The Strategic Rationale Model of the Actor Field Control Centre Coordinator
Figure 4.7: The Strategic Rationale Model of the Actor Community
link connecting the sub-goal to the two alternative means). Observe that both task-decomposition and means-ends links answer the “how” question relating to tasks and goals respectively. An SR model thus provides a means for modelling stakeholder interests, how they might be met, and the stakeholders’ evaluation of various alternatives with respect to their interests.

4.5 Mapping a Specific $i^*$ model into Z: Flood Rescue Management Case Study

The next step in our methodology is the mapping of specific $i^*$ model into Z schemas. Following steps are carried out to realise this goal:

1. Names of all the actors and external dependencies are specified. This is the first step in mapping the SD model of the Flood Rescue Management Case Study.

2. The second step in the mapping is based on the creation of Z schema for every dependency using $SDependency$ schema as a basis.

3. The first step in mapping the SR model is to specify the names of all the internal intentional elements of the selected actor.

4. The second step is the creation of a Z schema for every internal intentional element using $AElement$ schema as a basis.

Schemas for actors, dependencies, actor internal intentional elements and the links between them in a specific $i^*$ model are defined using these model schemas - we shall call these as element schemas.

4.5.1 Terminology used

All the actors and dependencies are specified by unique names. For example actor $FCCC$ is specified as field_coordinator, $ECCC$ as coordinator, $Volunteers/Emergency Workers$ as volunteer. Similarly, $Respond Quickly$ dependency corresponds to $respond_quickly$ in Z schema, $Weather Data$ dependency corresponds to $weather_data$ in Z schema. It is intuitively obvious because of the similarity of names (we use this similarity only for clarity purpose). Terminology used is specified in Table 4.2 as ready reference.
### Table 4.2: Terminology used in arriving at Z schemas

<table>
<thead>
<tr>
<th>Name of the actor/dependency</th>
<th>Specification in Z schemas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volunteers/Emergency Workers</td>
<td>workers</td>
</tr>
<tr>
<td>CallTaking Supervisor/System</td>
<td>supervisor</td>
</tr>
<tr>
<td>Field Control Centre Coordinator</td>
<td>field_coordinator</td>
</tr>
<tr>
<td>Emergency Coordination Centre Coordinator</td>
<td>coordinator</td>
</tr>
<tr>
<td>Community</td>
<td>community</td>
</tr>
<tr>
<td>Weather Bureau</td>
<td>bureau</td>
</tr>
<tr>
<td>Respond Quickly</td>
<td>respond_quickly</td>
</tr>
<tr>
<td>Rescue Assessment</td>
<td>rescue_assessment</td>
</tr>
<tr>
<td>Prioritised List of Jobs</td>
<td>prioritised_list_of_jobs</td>
</tr>
<tr>
<td>Local Info Update</td>
<td>local_info_update</td>
</tr>
<tr>
<td>Rescue Plan</td>
<td>rescue_plan</td>
</tr>
<tr>
<td>Weather Data</td>
<td>weather_data</td>
</tr>
<tr>
<td>Field Info</td>
<td>field_info</td>
</tr>
<tr>
<td>Evacuation</td>
<td>evacuation</td>
</tr>
<tr>
<td>Classifying Calls</td>
<td>classifying_calls</td>
</tr>
<tr>
<td>Fast Response</td>
<td>fast_response</td>
</tr>
<tr>
<td>Forward Calls</td>
<td>forward_calls</td>
</tr>
<tr>
<td>Answer Calls</td>
<td>answer_calls</td>
</tr>
<tr>
<td>Manage Emergency Calls</td>
<td>manage_emergency_calls</td>
</tr>
<tr>
<td>Mobile</td>
<td>mobile</td>
</tr>
<tr>
<td>Email</td>
<td>email</td>
</tr>
<tr>
<td>Manage Rescue Operation</td>
<td>manage_rescue_operation</td>
</tr>
<tr>
<td>Report Local Situation</td>
<td>report_local_situation</td>
</tr>
<tr>
<td>Manage Resources</td>
<td>manage_resources</td>
</tr>
<tr>
<td>Publicise Evacuation Routes</td>
<td>publicise_evacuation_routes</td>
</tr>
<tr>
<td>Use Loudspeakers</td>
<td>use_loudspeakers</td>
</tr>
<tr>
<td>Radio Transmission</td>
<td>radio_transmission</td>
</tr>
<tr>
<td>Asses Weather Situation</td>
<td>asses_weather_situation</td>
</tr>
<tr>
<td>Quick Efficiently</td>
<td>quick_efficiently</td>
</tr>
<tr>
<td>Publicise Evac Routes</td>
<td>publicise_evac_routes</td>
</tr>
<tr>
<td>Inform People at Risk</td>
<td>inform_people_risk</td>
</tr>
<tr>
<td>Manage Rescue Operation</td>
<td>manage_rescue_operation</td>
</tr>
<tr>
<td>Manage Calls</td>
<td>manage_calls</td>
</tr>
<tr>
<td>Report Local Situation</td>
<td>report_local_situation</td>
</tr>
<tr>
<td>Manage Resources</td>
<td>manage_resources</td>
</tr>
<tr>
<td>Manage Volunteers</td>
<td>manage_volunteers</td>
</tr>
<tr>
<td>Use Loudspeakers</td>
<td>use_loudspeakers</td>
</tr>
<tr>
<td>Analysed Weather Data</td>
<td>analysed_weather_data</td>
</tr>
<tr>
<td>Evacuation Routes</td>
<td>evacuation_routes</td>
</tr>
<tr>
<td>Activate Local Evacuation Plan</td>
<td>activate_local_evacuation_plan</td>
</tr>
</tbody>
</table>
4.5.2 Mapping the SD model

Names of all the actors and external dependencies are specified. This is the first step of mapping SD model of managing flood rescue and evacuation operations case study. First of all it is necessary to describe their names in Z using the following axiomatic definition:

\[
\text{coordinator, bureau, supervisor, volunteer, field\_coordinator, community : NAME}
\]
\[
\text{rescue\_plan, weather\_data, respond\_quickly, evacuation, field\_info}
\]
\[
\text{rescue\_assessment, prioritised\_list\_of\_jobs, local\_info\_update : NAME}
\]
\[
\text{all\_actors = \{} \text{coordinator, bureau, supervisor, volunteer, field\_coordinator, community} \text{\}}
\]
\[
\text{all\_depend = \{} \text{rescue\_plan, weather\_data, respond\_quickly, evacuation, field\_info, rescue\_assessment, prioritised\_list\_of\_jobs, local\_info\_update} \text{\}}
\]

We have specified only 8 out of 38 dependencies here but for appropriate mapping it is compulsory to specify all the names.

The second step in mapping is based on the creation of Z schema for every dependency using SDependencySR schema as a basis. For SD model of managing flood rescue and evacuation operations we need to create 38 schemas. Consequently, the degree of using Z notation corresponds with the complexity of the SD model. However, all Z schemas are of the same type, the process of their creation is simple and can be easily automated. As illustration, we are creating only 4 schemas for different types of dependencies below.

\[
\text{WeatherData}
\]
\[
\text{SDependency}
\]
\[
\text{dependum = weather\_data}
\]
\[
\text{dependuer = coordinator}
\]
\[
\text{depennee = bureau}
\]
\[
\text{dependuer\_internal\_element = analyse\_weather\_forecast}
\]
\[
\text{depennee\_internal\_element = forward\_weather\_data}
\]
\[
\text{type = resource}
\]
\[
\text{degree = committed}
\]
4.5. Mapping a Specific i* model into Z: Flood Rescue Management Case Study

<table>
<thead>
<tr>
<th>RescuePlan</th>
<th>SDependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependum = rescue_plan</td>
<td></td>
</tr>
<tr>
<td>depender = coordinator</td>
<td></td>
</tr>
<tr>
<td>dependee = field_coordinator</td>
<td></td>
</tr>
<tr>
<td>depender_internal_element = activate_emergency_plan</td>
<td></td>
</tr>
<tr>
<td>dependee_internal_element = activate_local_evacuation_plan</td>
<td></td>
</tr>
<tr>
<td>type = task</td>
<td></td>
</tr>
<tr>
<td>degree = committed</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RespondQuickly</th>
<th>SDependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependum = respond_quickly</td>
<td></td>
</tr>
<tr>
<td>depender = field_coordinator</td>
<td></td>
</tr>
<tr>
<td>dependee = volunteer</td>
<td></td>
</tr>
<tr>
<td>depender_internal_element = manage_volunteers</td>
<td></td>
</tr>
<tr>
<td>dependee_internal_element = rescue_people</td>
<td></td>
</tr>
<tr>
<td>type = softgoal</td>
<td></td>
</tr>
<tr>
<td>degree = committed</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evacuation</th>
<th>SDependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependum = evacuation</td>
<td></td>
</tr>
<tr>
<td>depender = community</td>
<td></td>
</tr>
<tr>
<td>dependee = supervisor</td>
<td></td>
</tr>
<tr>
<td>depender_internal_element = need_to_be_rescued</td>
<td></td>
</tr>
<tr>
<td>dependee_internal_element = answer_calls</td>
<td></td>
</tr>
<tr>
<td>type = goal</td>
<td></td>
</tr>
<tr>
<td>degree = committed</td>
<td></td>
</tr>
</tbody>
</table>

Line \(\text{dependum} = \text{evacuation}\) shows the name of the dependency. It is a \textit{goal} dependency so \textit{type} = \textit{goal}. The community actor depends on the supervisor so \textit{depender} = \textit{community} and \textit{dependee} = \textit{supervisor}. This schema also includes the names of the internal intentional elements (\textit{depender\_internal\_element} and \textit{dependee\_internal\_element}) linked to the dependency. The elements are \textit{need\_to\_be\_rescued} and \textit{answer\_calls} respectively. The importance of the dependency is not marked in the SD diagram hence we consider \textit{degree} = \textit{committed}.

Thus, \textit{RespondQuickly} schema corresponds to the \textit{respond\_quickly} dependency. It is intuitively obvious because of the similarity of names (we use this similarity only for clarity purpose). The formal correspondence between schemas and dependencies
is established by using variable \textit{dependum} inside the schemas without explicitly using the names of schemas. The formal rule of correspondence is described below:

\[
\begin{align*}
\text{correspond} : & \quad \text{NAME \mapsto SDependency} \\
\text{dom correspond} = & \quad \text{all\_depend} \\
\forall x : & \quad \text{NAME} \mid x \in \text{all\_depend} \bullet \\
& \quad (\text{correspond}(x)).\text{dependum} = x
\end{align*}
\]

The schemas for all the other dependencies are similar to the \textit{Evacuation} schema. Other Schemas can be realised in the same manner.

\subsection{Mapping the SR model}

The first step of mapping (formalisation) SR model of managing flood rescue and evacuation operations case study is the creation of a Z schema for every actor using Actor schema as a starting point. In these schemas we need to specify names of all the internal intentional elements of the selected actor. The schemas for supervisor and field\_coordinator actors are provided as ready reference below:

\begin{verbatim}
Supervisor
Actor
classifying\_calls, fast\_response, mobile, email, forward\_calls,
manage\_emergency\_calls, answer\_calls : NAME

actor\_name = supervisor
actor\_element = {classifying\_calls, fast\_response, mobile, email,
forward\_calls, manage\_emergency\_calls, answer\_calls}
\end{verbatim}

\begin{verbatim}
FieldCoordinator
Actor
manage\_rescue\_operation,
report\_local\_situation, manage\_resources,
activate\_local\_evacuation\_plan, publicise\_evacuation\_routes, 
manage\_volunteers, use\_loudspeakers, radio\_transmission, 
asses\_weather\_situation, quick\_efficiently, plan\_rescue : NAME

actor\_name = field\_coordinator
actor\_element = {manage\_rescue\_operation, 
report\_local\_situation, manage\_resources, 
activate\_local\_evacuation\_plan, publicise\_evacuation\_routes, 
manage\_volunteers, use\_loudspeakers, radio\_transmission, 
asses\_weather\_situation, quick\_efficiently, plan\_rescue}
\end{verbatim}
The second step is the creation of a Z schema for every internal intentional element using $AElement$ schema as a basis. As an example, we will need to create 7 schemas for the internal intentional elements of $supervisor$ actor. It is necessary to specify the name of the dependee, the type and the degree of the element (similar to external dependencies) but also the kind of the link and names of external and internal components of the considered element. To demonstrate this approach, we are showing five schemas as ready reference:

<table>
<thead>
<tr>
<th>Schema</th>
<th>AElement</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FastResponse</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dependum</td>
<td>$fast_response$</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>$softgoal$</td>
<td></td>
</tr>
<tr>
<td>degree</td>
<td>$committed$</td>
<td></td>
</tr>
<tr>
<td>int_components</td>
<td>${mobile, email}$</td>
<td></td>
</tr>
<tr>
<td>contrib_p</td>
<td>${mobile}$</td>
<td></td>
</tr>
<tr>
<td>contrib_n</td>
<td>${email}$</td>
<td></td>
</tr>
<tr>
<td>ext_components</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td>contrib</td>
<td></td>
</tr>
</tbody>
</table>

| ManageCalls     |          |                             |
| dependum        | $manage\_calls$ |
| type            | $task$    |
| degree          | $committed$     |
| int_components  | $\{classifying\_calls, forward\_calls, fast\_response, answer\_calls\}$ |
| ext_components  | $\{inform\_people\_risk, plan\_rescue, list\_people\_tobe\_rescued\_1, quick\_response\_1, list\_people\_tobe\_rescued\_2, quick\_response\_2\}$ |
| link            | task\_decomp   |

| ClassifyingCalls|          |                             |
| dependum        | $classifying\_calls$ |
| type            | $task$    |
| degree          | $committed$     |
| int_components  | $\emptyset$      |
| ext_components  | $\emptyset$      |
| link            | $\emptyset$      |
| link            | NA            |
4.6. Refinement

We have considered Z schemas represented above as part of one to one mapping of $i^*$ models into the Z notation. Using this approach, all the information from the $i^*$ models is reflected in Z. We shall refer to these basic schemas as model schemas. Schemas for actors, dependencies, actor internal elements and the links between them in a specific $i^*$ model are defined using these model schemas - we shall call these element schemas. The mapping process that we have described so far leads to a Z specification that captures the structure represented in an $i^*$ model (and in the instance of states, obliges the analyst to represent some additional information as well). A key subsequent step is the refinement of these essentially structural schemas with additional information (i.e. information not included in an $i^*$ model, but obtained via further analysis-e.g., temporal sequencing of dependencies, fulfillment conditions for dependencies etc). The finer points of this refinement are provided in the next section. We shall refer to the Z specification obtained after these refinements as the Extended Z Specification.

4.6 Refinement

We feel that it is not feasible to capture and perceive complete information from the $i^*$ model and represent it in the form of Z schemas, the reason being that some information
is always concealed in the $i^*$ model which depends on the experience of the analyst to present it to the world. The mapping of $i^*$ models into Z schemas enables the refinement of these schemas with additional information, such as invariant properties, fulfillment conditions etc. (Note that these cannot be represented in the original $i^*$ model). We felt the need to refine this process of mapping $i^*$ diagram into corresponding Z schemas by providing some important refinement steps. We are providing the essence of some possible approaches with examples.

1. **Order or Sequence on the dependencies**: When we inspect SD and SR diagrams, we are not in a position to ascertain the order or sequence in which various dependencies are realised. This missing information can be easily represented in the resulting Z schemas without making any changes to the SD or SR diagrams.

   **EXAMPLE I**: Refer to the SR diagram which describes the intentional relationships that are “internal” to the \textit{CallTakingSupervisor/System} actor. It is necessary to answer and classify a call before forwarding it to the appropriate authority so dependencies \textit{answer calls} and \textit{classifying calls} should be realised before dependency \textit{forward calls}. This fact is not reflected in the $i^*$ diagrams but can be easily incorporated into Z schemas. For this, it is necessary to include into the predicate part of \textit{ForwardCalls} schema the following precondition:

   \[
   (SD\text{state}(\text{forward\_calls}) = \text{fulfilled}) \Rightarrow ((SD\text{state}(\text{answer\_calls}) = \text{fulfilled}) \land (SD\text{state}(\text{classifying\_calls}) = \text{fulfilled})).
   \]

   \[
   \begin{array}{|c|}
   \hline
   \text{ForwardCalls} \\
   \hline
   \text{AE}\text{Element} \\
   \text{Supervisor} \\
   \hline
   \text{dependum} = \text{forward\_calls} \\
   \text{type} = \text{goal} \\
   \text{degree} = \text{committed} \\
   \text{int\_components} = \{\text{mobile, email}\} \\
   \text{ext\_components} = \emptyset \\
   \text{link} = \text{means\_ends} \\
   (SD\text{state}(\text{forward\_calls}) = \text{fulfilled}) \Rightarrow ((SD\text{state}(\text{answer\_calls}) = \text{fulfilled}) \land (SD\text{state}(\text{classifying\_calls}) = \text{fulfilled})).
   \end{array}
   \]

   **EXAMPLE - II**: For realising resource dependency \textit{PrioritisedListofJobs}, \textit{Volunteers/EmergencyWorkers} agent depends on the \textit{FieldControlCentreCoordinator} agent. Prior to the success of this dependency, following dependencies must be realised; Task dependency \textit{contact\_emergency\_services} between \textit{Community} and
the CallTakingSupervisor/System agents and Resource dependency
list_location_people_tobe_rescued between CallTakingSupervisor/System and Field-
ControlCentreCoordinator agents. For this, it is necessary to include into the
predicate part of PrioritiseList schema the following precondition:

\[
(SDstate(prioritised_list_jobs) = \text{fulfilled}) \Rightarrow
((SDstate(contact\_emergency\_services) = \text{fulfilled}) \land
(SDstate(list_location_people_tobe_rescued) = \text{fulfilled})).
\]

2. Resource Dependency Structure: After inspecting the \(i^*\) diagrams, we cannot
predict about the organisation, construction and contents of the given resource
dependency. We shall now present an example to explain different categories of
resources possible in the \(i^*\) framework with their contents and mode of commu-
nication.

Free types CONTENT\_TYPE, FORM\_TYPE describe the contents and form (or
mode of communication) of the resources respectively.

\[
\text{CONTENT\_TYPE ::= yes\_no | running\_text | list\_information} \\
| \text{material\_equipment | technical\_data} \\
| \text{miscellaneous}
\]

\[
\text{FORM\_TYPE ::= e\_mail | fax | booklets | leaflets | telephone} \\
| \text{mobile\_phone | pager | miscellaneous}
\]

EXAMPLE - I: The FieldControlCentreCoordinator actor depends on the EmergencyCoordinationCentreCoordinator actor for providing AnalysedWeatherData.
This is modelled as a resource dependency in the SD diagram. The contents of
this resource are information about the flood warnings, rainfall (rainfall location
and intensity - to be forwarded every 10 minutes) and river information (if any in the area), forecast and observation index etc. This resource is in the form of technical data/report and is communicated via e-mail attachment or faxed.

```plaintext
AnalysedData

SDependency
content : CONTENT_TYPE
form : FORM_TYPE

dependum = analysed_weather_data
depend = field_coordinator
dependee = coordinator
depend internal element = assess_weather_situation
dependee internal element = analyse_weather_forecast
type = resource
degree = committed
content = technical_data
form = e_mail \ fax
```

**EXAMPLE - II:** The FieldControlCentreCoordinator actor depends on the Call-TakingSupervisor/System actor for forwarding the List&LocationofPeopletobeRescued. This is modelled as a resource dependency in the SD diagram. It consists of the list of people to be rescued along with their address, location and telephone numbers. There is priority on the list depending upon the flood/storm situation. This resource is in the form of List and is communicated via e-mail, fax or mobile/radio communication.

```plaintext
ListTobeRescued

SDependency
content : CONTENT_TYPE
form : FORM_TYPE

dependum = list_people_tobe_rescued
depend = field_coordinator
dependee = supervisor
depend internal element = manage_rescue_operation
dependee internal element = manage_emergency_calls
type = resource
degree = committed
content = list_information
form = e_mail \ mobile_phone \ fax
```

3. Association or Connection between different resources: The information about the association between the resources is missing in the i* framework. Whether
two or more resource dependencies are connected with each other in a certain way is missing in the SD and SR diagrams. Some resources may be subset of the preceding resource dependency, on the other hand some resource dependencies might contain the same information (but they might exist between different actors). We shall now present an example to explain this refinement case.

**EXAMPLE:** The resource dependency *PrioritisedListofJobs* (between *Volunteers/EmergencyWorker* agent and *FieldControlCentreCoordinator*) contains all the information from the resource dependency *LocationofPeopletoBeRescued* (between *CallTakingSupervisor/System* agent and *FieldControlCentreCoordinator*). We can say that first resource dependency is the subset or part of the second resource dependency. The other observation is that the second resource precedes the first resource dependency. We are going to use the basic type *RESCUE_INFORMATION* to describe the information about the people to be rescued.

```
[RESCUE_INFORMATION]
```

```
PrioritiseListJobs
SDependency
list_jobs : P RESCUE_INFORMATION

dependum = list_jobs
depender = volunteer
dependee = field_coordinator
depender_internal_element = prepare_for_rescue
dependee_internal_element = manage_rescue_operation
type = resource
degree = committed
∀ ListPeopleTobeRescued •
   ListPeopleTobeRescued.list_people ⊆ list_jobs
SD_state(prioritised_list_jobs) = fulfilled
⇒ SD_state(list_people_tobe_rescued) = fulfilled
```

4. **Temporal features and operators:** Temporal Logic allows us to specify assertions about program behavior as time progresses. In other words we can describe sequences of state changes and properties of behaviors. We observe that it is worthwhile to introduce temporal logic features and operators in the mapping from $i^*$ model to Z schemas.

The state function $SD\_state$ represents the snapshot state of the system. To
describe the behaviour of the system in time, consider all the possible sequences of system states

\[
\begin{align*}
\text{SDscenarios} & : \mathcal{P}(\text{seq SD}) \\
\text{SDfuture}, \text{SDpast} & : \text{SD} \rightarrow \mathcal{P}(\text{seq SD}) \\
\text{ran SDfuture} \cup \text{ran SDpast} & \subseteq \mathcal{P} \text{ SDscenarios} \\
\forall s : \text{SD} \cdot \text{SDfuture}(s) & = \{f : \text{seq SD} \mid \text{head } f = s\} \land \\
\text{SDpast}(s) & = \{p : \text{seq SD} \mid \text{last } p = s\}
\end{align*}
\]

For each state function \( s \) consider all the behaviours \( \text{SDfuture} \) which are started in \( s \) (the future of the system) and behaviours \( \text{SDpast} \) that are finished in \( s \) (the past of the system). The operators \( \Box \phi \) \text{ always in the future}, \( \circ \phi \) \text{ next state}, \( \Diamond \phi \) \text{ eventually in the future} are temporal operators to perform reasoning in time. These are standard operators in the typed-linear temporal logic language (LTL) [55]. LTL supports the standard boolean and relational operators. The logic also provides universal (\( \forall \)) and existential (\( \exists \)) quantifiers, which range over all the instances of a given class, and a set of future and past temporal operators.

We can formalise all the main temporal operators such as \textit{sometimes in the past}, \textit{always in the past}, \textit{sometimes in the future}, \textit{always in the future}, etc., which are used in different techniques of requirements engineering, for example, KAOS [33], Formal Tropos [55]. Thus, the operator \( \Box \phi \) \textit{ always in the future} [55] for state \( s \) can be modelled as:

\[
\forall c : \text{seq SD}; \ st : \text{SD} \mid c \in \text{SDfuture}(s) \land \langle st \rangle \subseteq c \cdot \phi
\]

Correspondingly, the operator \( \circ \phi \) \textit{next state} for state \( s \) can be modelled as

\[
\forall c : \text{seq SD}; \ st : \text{SD} \mid c \in \text{SDfuture}(s) \land st = c(2) \cdot \phi
\]

and the operator \( \Diamond \phi \) \textit{eventually in the future} for state \( s \) can be modelled as

\[
\forall c : \text{seq SD} \mid c \in \text{SDfuture}(s) \cdot \exists st : \text{SD} \mid \langle st \rangle \subseteq c \cdot \phi
\]

For dealing with systems which stipulate special timing requirements (like concurrent real-time reactive systems), it is possible to use special extensions of Z like Timed Communicating Object Z (TCOZ) [90] which are designed for modelling real-time applications.
4.7 Summary

This chapter presented our proposal that Z formal notation and the $i^*$ modelling framework can function in a complementary and synergistic fashion. This approach makes use of the advantages of $i^*$ for the early-phase of requirements engineering and then continues with the specification of requirements in Z. The Z notation permits us to specify requirements with a degree of precision and formality that $i^*$ does not. The mapping process that we have described so far leads to a Z specification that captures the structure represented in an $i^*$ model (and in the instance of states, obliges the analyst to represent some additional information as well). A key subsequent step shown in this chapter is the refinement of these essentially structural schemas with additional information. We shall refer to the Z specification obtained after these refinements as the Extended Z Specification.

In the next chapter, we shall introduce a methodology for supporting the co-evolution of $i^*$ models and Z specifications. This chapter also explores how consistency is preserved during the co-evolution of $i^*$ models and Z specifications.
Chapter 5

Co-evolution of Formal Specifications and Agent-Oriented Conceptual Models

In this chapter, we shall introduce a methodology supporting the co-evolution of \(i^*\) models and Z specifications. The proposed methodology permits the maintenance of loose coupling between an \(i^*\) model and Z specification. The issue of preserving consistency in the co-evolution of formal and informal models is also discussed in this chapter.

5.1 Introduction

As we have pointed out earlier that, number of proposals have been made for combining \(i^*\) modelling with late-phase requirements analysis and the downstream stages of the software life-cycle. The Tropos project [24] uses the \(i^*\) notation to represent early- and late-phase requirements, architectures and detailed designs (Figure 5.1). However, the \(i^*\) notation in itself is not expressive enough to represent late-phase requirements, architectures and designs. To address this problem, a custom-designed formal language called Formal Tropos [55] has been proposed. Proposals to integrate \(i^*\) with formal agent programming languages have also been reported in the literature [61, 125, 145]. This chapter has similar objectives, but takes a somewhat different approach. We believe that the value of conceptual modelling in the \(i^*\) framework lies in its use as a notation complementary to existing specification languages, i.e., the expressive power of \(i^*\) complements that of existing notations. The use of \(i^*\) in this fashion requires that we define methodologies that support the co-evolution of \(i^*\) models with more traditional specifications. We use the notion of co-evolution in a very specific sense to describe a class of methodologies that permit \(i^*\) modelling to proceed independently of specification in a distinct notation, while maintaining some modicum of loose coupling via consistency constraints. In the current instance, we examine how this might be done with formal specification notations, but such an exercise is of value in the context...
5.1. Introduction

When proposing the co-evolution of two otherwise disparate approaches for requirements engineering, we need to take care the issue of maintaining consistency between the two approaches. The mapping rules can be viewed as providing formal semantics to $i^*$ diagrams by mapping this notation into Z specifications, a language which has well-defined formal semantics. A set of mapping rules is defined to help ensure consistency between the two models.

Consider the following example from our earlier case study (ongoing from the last chapter), which concentrates on a key function of the Emergency Services Agency (ESA): Managing Flood Rescue and Evacuation operations (this research has been conducted in the context of a larger project to deploy $i^*$ for Enterprise Modelling in a large ESA). This case study will be used to illustrate the proposed methodology in this chapter as well.
Table 5.1: Key steps in the proposed methodology of co-evolution of formal and informal requirements

<table>
<thead>
<tr>
<th>Key steps in the proposed methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Define techniques for reflecting changes in an i* model in the corresponding (unrefined) Z specification</td>
</tr>
<tr>
<td>(i). These are the addition and deletion, respectively, of the following eight elements:</td>
</tr>
<tr>
<td>- Addition/deletion of a dependency to an existing SD model.</td>
</tr>
<tr>
<td>- Addition/deletion of a task to an existing SR model.</td>
</tr>
<tr>
<td>- Addition/deletion of a goal to an existing SR model.</td>
</tr>
<tr>
<td>- Addition/deletion of a resource to an existing SR model.</td>
</tr>
<tr>
<td>- Addition/deletion of a softgoal to an existing SR model.</td>
</tr>
<tr>
<td>- Addition/deletion of a means-ends link to an existing SR model.</td>
</tr>
<tr>
<td>- Addition/deletion of a task decomposition link to an existing SR model.</td>
</tr>
<tr>
<td>- Addition of an actor to an existing i* diagram.</td>
</tr>
<tr>
<td>2. Define techniques for reflecting the refinements contained in the prior extended Z specification to obtain a new extended Z specification</td>
</tr>
<tr>
<td>(i). Identify the set of Z schemas in the prior collection of (unrefined) Z schemas (obtained from the prior i* model) that were refined in some fashion.</td>
</tr>
<tr>
<td>(ii). Identify schemas with the same names (if they exist, since some might have been deleted) in the current collection of (unrefined) Z schemas (obtained from the revised i* model).</td>
</tr>
<tr>
<td>(iii). Apply the same refinements to these identified schemas. This gives us the new extended Z specification.</td>
</tr>
</tbody>
</table>
5.2 Methodology Supporting the Co-evolution of i* and Z

The focus of our work in this chapter is on defining a methodology that permits the maintenance of loose coupling between an i* model and a Z specification. Our strategy is to localise the impact of changes. We do this at two specific points (refer to Figure 5.2).

1. We define techniques for reflecting changes in an i* model in the corresponding (unrefined) Z specification (i.e., the Z model obtained by directly applying the mapping techniques discussed in the previous section to the prior i* model).

2. We define techniques for reflecting the refinements contained in the prior extended Z specification to obtain a new extended Z specification (i.e., one which contains all of the prior refinements, while reflecting the changes in the corresponding i* model). We note that changes in the i* model only affect the element schemas, but not the model schemas.

Let us consider the first of these two questions: obtaining an unrefined Z specification from the modified i* model. We define techniques for achieving this that require reference to the prior i* model and the corresponding prior unrefined Z specification. We note that sixteen categories of possible changes may occur to an i* model. These are the addition and deletion, respectively, of the following eight elements:

- Dependencies
- Tasks
5.2. Methodology Supporting the Co-evolution of $i^*$ and Z

Figure 5.2: Co-evolution of $i^*$ models and Z specifications
Goals

- Resources

- Softgoals

- Means-end links

- Task-decomposition links

- Actors

We shall consider each of these cases in turn.

**Addition/deletion of a dependency to an existing SD model:**

i) Addition leads to the creation of an additional element schema for the new dependency (deletion leads to the removal of this schema).

ii) The internal intentional elements as represented in the SR models for the pair of actors involved in the dependency may need to be modified, since all the external dependencies are connected to some internal element of an actor. This change is localised to the following simple step: we add (or delete) the dependency name from the ext_components set in the corresponding element schema for the relevant internal element.

**Addition/deletion of a task to an existing SR model:**

i) Addition will result in the creation of a new element schema for the task (deletion leads to its removal). A newly added task is typically related via a means-ends link to a goal, or via a task decomposition link to another task. Potentially, it may also be related via a softgoal contribution link to an existing softgoal. Schemas for these links must then also be added along the lines described below.

ii) The element schemas for the goals, tasks and softgoals that this new task might be linked to (as discussed above) need to be modified by adding (respectively deleting) the name of the task to the int_components set of the corresponding schema(s).

iii) The name of the task must be added (respectively deleted) to the actor_element set in the element schema for the corresponding actor.

iv) The name of the task must be added (respectively deleted) as the value of the depender_internal_element variable in the schema for any dependency related to the task (should such a relationship be established after the task is added) in which the corresponding actor (into whose SR model the task has been added) is the dependant. In a similar fashion, the name of the task is added as the value of the


5.2. Methodology Supporting the Co-evolution of \( i^* \) and \( Z \)

The \( \text{dependee}_\text{internal_element} \) variable in the schema of any dependency related to the task in which the corresponding actor is the dependee.

\( v \) A downstream effect of the addition of a task in an SR model followed by the creation of a new dependency connecting this task to an internal element in another actor is that the steps outlined for the addition (respectively deletion) of a dependency (outlined above) have to be followed.

**Addition/deletion of a goal to an existing SR model:**

\( i \) Addition will result in the creation of a new element schema for the goal (deletion leads to its removal). A newly added goal is typically related via a means-ends link to a task, or via a task decomposition link to another task. Schemas for these links must then also be added along the lines described below.

\( ii \) The element schemas for the tasks that this new goal might be linked to (as discussed above) need to be modified by adding (respectively deleting) the name of the goal to the \( \text{int}_\text{components} \) set of the corresponding schema(s).

\( iii \) The name of the goal must be added (respectively deleted) to the \( \text{actor}_\text{element} \) set in the element schema for the corresponding actor.

**Addition/deletion of a resource to an existing SR model:**

\( i \) Addition will result in the creation of a new element schema for the resource (deletion leads to its removal). A newly added resource is typically related via a means-ends link to a task, or via a task decomposition link to another task. Schemas for these links must then also be added along the lines described below.

\( ii \) The element schemas for the tasks that this new resource might be linked to (as discussed above) need to be modified by adding (respectively deleting) the name of the resource to the \( \text{int}_\text{components} \) set of the corresponding schema(s).

\( iii \) The name of the resource must be added (respectively deleted) to the \( \text{actor}_\text{element} \) set in the element schema for the corresponding actor.

**Addition/deletion of a softgoal to an existing SR model:**

\( i \) Addition will result in the creation of a new element schema for the softgoal (deletion leads to its removal). A newly added softgoal is typically related via a task decomposition link to another task. Potentially, it may also be related via a softgoal contribution link to an existing softgoal or a task. Schemas for these links must then also be added along the lines described below.

\( ii \) The element schemas for the tasks and softgoals that this new softgoal might be linked to (as discussed above) need to be modified by adding (respectively deleting) the name of the softgoal to the \( \text{int}_\text{components} \) set of the corresponding schema(s).
iii) The name of the softgoal must be added (respectively deleted) to the *actor_element* set in the element schema for the corresponding actor.

*Addition/deletion of a means-ends link to an existing SR model:*

Means-ends links (as with task decomposition links) are not represented via separate schemas, but via the schemas of the internal (SR) elements that they relate. A means-ends link offers alternative means for achieving a given goal (we shall refer to this as the *end*). In other words, it is effectively the analogue of an OR node in an AND-OR goal graph.

The addition of a means-ends link results in the value of the *link* variable in the element schema for the end being assigned the value *means-ends* and the *int_components* set in the same schema being defined as the collection of the internal SR elements (which could be tasks, goals or resources) related to the end via the means-ends link. Deletion results in these values being removed.

*Addition/deletion of a task decomposition link to an existing SR model:*

A task decomposition link functions as the analogue of an AND node in an AND-OR goal graph and provides a singly, unique means of decomposing a task (we shall refer to this as the *parent task*) into a collection of subtasks, subgoals, resources etc. The addition of a task decomposition link results in the following changes to the element schema for the parent task: the *link* variable is assigned the value *task-decomposition* while the *int_components* set is defined as the collection of subtasks, subgoals etc. related to the parent task by this link. Deletion results in these values being removed.

*Addition of an actor to an existing *i* diagram will lead to following four steps:*

A new element schema for the actor is created. In the instance of each internal (SR) element for the actor, the steps outlined above are followed. The same applies for any dependencies that this actor might participate in.

We shall now discuss the second area where we are able to localise the impact of changes: the generation of a new extended Z specification given the new set of Z schemas (corresponding to the modified *i* model) and the prior extended (refined) Z specification. Our aim is to reflect the refinements in prior set of Z schemas (that led to the prior extended Z specification) in the new collection of Z schemas, without having to re-do the refinements. This is a relatively simple affair. We identify the set of Z schemas in the prior collection of (unrefined) Z schemas (obtained from the prior *i* model) that were refined in some fashion. We identify schemas with the same names (if they exist, since some might have been deleted) in the current collection of (unrefined) Z schemas (obtained from the revised *i* model), and apply the same refinements to
5.2. Methodology Supporting the Co-evolution of \(i^*\) and \(Z\) these. This gives us the new extended \(Z\) specification.

We shall now present few illustrations to explain the methodology supporting the co-evolution of \(i^*\) and \(Z\). These examples are based on the managing flood rescue and evacuation case study. The case study shown in Figure’s 4.1 and 4.2 is used to illustrate the SD and SR models of managing flood rescue and evacuation operations by ESA. Following modifications/additions were performed on the initial \(i^*\) diagrams:

Introducing a resource dependency \(\text{SimplifiedWeatherData}\) between \(\text{Volunteers/EmergencyWorkers}\) and \(\text{FCCC}\) will lead to the modification of the original \(i^*\) diagram and creation of an additional element \(Z\) schema (external dependency). The concerned actors \(\text{Volunteers/EmergencyWorkers}\) and \(\text{FCCC}\) internal intentional elements \(Z\) schema(s) (which is affected) are going to be modified because of this action (since all the external dependencies are connected to some internal element of an actor somewhere in the SR diagram). In this case the internal intentional elements in \(\text{Volunteers/EmergencyWorkers}\) and \(\text{FCCC}\) are \(\text{RescuePeople}\) and \(\text{AssesWeatherSituation}\) respectively. The modified SD and SR models of the managing flood rescue and evacuation operations by ESA are shown in figure’s 5.3 and 5.4 respectively.

\[
\text{SimplifiedWeatherData}
\]

\[
\text{SDependency}
\]

\[
\begin{align*}
\text{dependum} &= \text{simplified\_weather\_data} \\
\text{dependee} &= \text{volunteer} \\
\text{dependee\_internal\_element} &= \text{field\_coordinator} \\
\text{dependee\_internal\_element} &= \text{rescue\_people} \\
\text{dependee\_internal\_element} &= \text{asses\_weather\_situation} \\
\text{type} &= \text{resource} \\
\text{degree} &= \text{committed}
\end{align*}
\]

Line \(\text{dependum} = \text{simplified\_weather\_data}\) shows the name of the dependency. It is a resource dependency so \(\text{type} = \text{resource}\). The volunteer actor depends on the \(\text{field\_coordinator}\) so \(\text{dependee} = \text{volunteer}\) and \(\text{dependee} = \text{field\_coordinator}\). This schema also includes the names of the internal intentional elements (\(\text{dependee\_internal\_element}\) and \(\text{dependee\_internal\_element}\)) linked to the dependency. The elements are \(\text{rescue\_people}\) and \(\text{asses\_weather\_situation}\) respectively. The importance of the dependency is not marked in the SD diagram hence we consider \(\text{degree} = \text{committed}\).

The newly added resource dependency \(\text{SimplifiedWeatherData}\)’s, \(Z\) schema is further refined with additional information derived from the \(i^*\) models (refinement)-this is known as Extended \(Z\) model. Our observation is that this extended \(Z\) schema
5.2. Methodology Supporting the Co-evolution of $i^*$ and $Z$

Figure 5.3: Modified Strategic Dependency Model of the Flood Rescue Management case study
5.2. Methodology Supporting the Co-evolution of $i^*$ and $Z$

Figure 5.4: Modified Strategic Rationale Model of the Flood Rescue Management case study
is not going to affect any other previous extended Z schema. We can directly perform some minor modifications in the predicate part of the newly created Z schema of the resource dependency (as basis) to arrive at this extended Z schema. For example, dependency `analysed_weather_forecast` should be realised before dependency `simplified_weather_data`. For this, it is necessary to include into the predicate part of `SimplifiedData` schema the following precondition:

\[
SD\_state(simplified\_weather\_data) = \text{fulfilled} \Rightarrow SD\_state(\text{analysed}\_\text{weather}\_\text{forecast}) = \text{fulfilled}.
\]

\[
\begin{align*}
\text{SimplifiedData}_1 & \quad S\text{Dependency} \\
\text{dependum} = \text{simplified\_weather\_data} & \quad \text{dependee} = \text{field\_coordinator} \\
\text{depende} = \text{volunteer} & \quad \text{dependee\_internal\_element} = \text{rescue\_people} \\
\text{dependee\_internal\_element} = \text{asses\_weather\_situation} & \quad \text{type} = \text{resource} \\
\text{degree} = \text{committed} & \\
SD\_state(simplified\_weather\_data) = \text{fulfilled} & \Rightarrow SD\_state(\text{analysed}\_\text{weather}\_\text{forecast}) \\
& = \text{fulfilled}
\end{align*}
\]

Based on the The revised Z schemas of internal intentional elements `Rescue People` and `AssesWeatherSituation` are going to have `SimplifiedWeatherData` as additional entry under the ext_components set in respective Z schemas. Initial and the revised Z schemas of the internal intentional elements are provided as ready reference:

\[
\begin{align*}
\text{AssesWeatherSituation} & \quad A\text{Element} \\
\text{Field\_coordinator} & \\
\text{dependum} = \text{asses\_weather\_situation} & \quad \text{type} = \text{task} \\
\text{degree} = \text{committed} & \\
\text{int\_components} = \emptyset & \\
\text{ext\_components} = \{\text{analysed\_weather\_forecast}\} & \\
\text{link} = \text{NA}
\end{align*}
\]
The rest of the mapped Z schemas remain unchanged for the modified i* model.
5.3. Preserving Consistency in the Co-evolution of Formal and Informal Models

We note that a reverse mapping from a collection of Z schemas to an \( i^* \) model is possible provided the following assumptions hold.

- The Z schemas were obtained from an initial \( i^* \) model via mapping and refinement along the lines described above
- The prior \( i^* \) model is available for reference
- The integrity of the element schemas must be maintained throughout the refinement process, i.e., refinement steps may add to but not modify existing element schemas.

Given these assumptions it is relatively simple to identify the named element schemas in a Z specification and thus reconstruct the corresponding \( i^* \) model without loss of information (any refinements made will, of course, not be reflected in the \( i^* \) model).

5.3 Preserving Consistency in the Co-evolution of Formal and Informal Models

When proposing the co-evolution of two otherwise disparate approaches for requirements engineering, we need to maintain consistency between the two approaches. The mapping rules can be viewed as providing formal semantics to the \( i^* \) diagrams by mapping this notation into Z specifications, a language which already has rich formal semantics. We believe that these semantics are largely consistent with the somewhat implicit semantics for \( i^* \) developed in [153]. A set of mapping rules is defined to help ensure consistency between the two models. We have proposed a set of mapping rules that constrains the modeler to map the elements of the \( i^* \) model to appropriate Z schemas and ensures that the two models are consistent. This allows us to trace corresponding elements in the two models when changes are made. We are interested in providing a taxonomy of inconsistencies that may occur from translating \( i^* \) models into Z specifications (and their co-evolution). The main types of inconsistencies that may occur when performing the co-evolution of formal and informal models are listed below. The discussion on how our methodology provides support to overcome these issues is presented.

*Structural inconsistency:* According to our methodology, it is necessary to introduce Z schemas corresponding to the elements in the \( i^* \) model. If the Z specification lacks a
schema for a certain $i^*$ element, the combined model is inconsistent with respect to this regime. In our co-evolution methodology we are keeping the structural inconsistency issue under control by strictly adhering to the mapping rules to accommodate any changes. This allows us to keep track of corresponding elements in the two models when changes are made. The mapping process that we have described so far leads to a Z specification that captures the structure represented in an $i^*$ model (and in the instance of states, obliges the analyst to represent some additional information as well). Hence, parsing of Z specifications will lead to one $i^*$ model. Likewise, from the given $i^*$ model we are in a position to arrive at Z specifications which capture and represent all the structural information contained in the given $i^*$ model. Hence, with the help of clear mapping rules and a supporting methodology we are in a position to avoid structural inconsistencies.

**Semantic inconsistency:** As we have explained earlier, the mapping rules can be viewed as giving a formal semantics to $i^*$ diagrams by mapping this notation into Z specifications, a language which has well-defined formal semantics. Semantic inconsistencies may arise if the creation conditions are contradictory; invariants are not maintained. Inconsistencies may arise if the default creation condition of a subgoal of a task decomposition link or a means-ends link is that the parent goal exists, but has not been fulfilled. The fulfillment condition of the parent goal depends on the fulfillment of the subgoals. If the subgoals are connected to the parent goal with means-ends links, then fulfillment of at least one of the subgoals is necessary for the fulfillment of the parent goal. If they are connected with task-decomposition links then the fulfillment of all the subgoals is necessary. We have proposed a set of translation rules and guidelines that permit us to systematically derive these constraints. These rules capture the intuitive semantics that we use when designing an $i^*$ model. For instance, a temporal ordering or sequencing refinement technique is applied in the Z schema of the parent task in the task decomposition links to include the pre-condition that all of the subgoals or subtasks are fulfilled prior to the fulfillment of the parent task. This helps us in taking care of semantic inconsistencies which may arise in the mapping of $i^*$ diagrams into Z specifications.

Semantic inconsistencies can be avoided in the case of task-decomposition links in $i^*$ model if all the subgoals, subtasks or softgoals connected to the parent task are realised (fulfilled or satisficed) before the realisation of the parent task. We shall provide an example to explain this. We observe that task dependency *manage_calls* (for Supervisor agent) is decomposed into subtasks *classifying_calls* and *answer_calls*, subgoal
5.3. Preserving Consistency in the Co-evolution of Formal and Informal Models

forward_calls and softgoal fast_response. To avoid semantic inconsistencies, we should include in the predicate part of the Z schema of the task dependency manage_calls, following precondition:

\[(SDstate(manage_calls) = fulfilled) \Rightarrow ((SDstate(answer_calls) = fulfilled) \land (SDstate(classifying_calls) = fulfilled) \land (SDstate(fast_response) = satisfied) \land (SDstate(forward_calls) = fulfilled))\]

<table>
<thead>
<tr>
<th>ManageCalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>AElement</td>
</tr>
<tr>
<td>Supervisor</td>
</tr>
</tbody>
</table>

\[\text{dependum} = \text{manage\_calls}\]
\[\text{type} = \text{task}\]
\[\text{degree} = \text{committed}\]
\[\text{int\_components} = \{\text{classifying\_calls, forward\_calls, fast\_response, answer\_calls}\}\]
\[\text{ext\_components} = \{\text{inform\_people\_risk, plan\_rescue, list\_people\_tobe\_rescued\_1, quick\_response\_1, list\_people\_tobe\_rescued\_2, quick\_response\_2}\}\]
\[\text{link} = \text{task\_decomp}\]
\[(SDstate(manage_calls) = fulfilled) \Rightarrow ((SDstate(answer_calls) = fulfilled) \land (SDstate(classifying_calls) = fulfilled) \land (SDstate(fast_response) = satisfied) \land (SDstate(forward_calls) = fulfilled))\]

The same kind of reasoning can be performed for the means-ends link in \(i^*\) model using Z. Let us consider the case of a goal dependency named forward_calls (for Supervisor agent). Here, the subtasks mobile, email are connected to the parent goal with the means-ends links. Fulfillment of at least one of the subtasks is necessary for the fulfillment of the parent goal forward_calls. To avoid semantic inconsistency, we should include in the predicate part of the Z schema of the goal dependency forward_calls, following precondition:

\[(SDstate(forward_calls) = fulfilled) \Rightarrow ((SDstate(mobile) = fulfilled) \lor (SDstate(email) = fulfilled))\]
5.3. Preserving Consistency in the Co-evolution of Formal and Informal Models

ForwardCalls
AElement
Supervisor
dependum = forward_calls
type = goal
degree = committed
int_components = \{mobile, email\}
ext_components = ∅
link = means_ends
(SDstate(forward_calls) = fulfilled) ⇒
((SDstate(mobile) = fulfilled) ∨
(SDstate(email) = fulfilled))

State invariants are associated with the state variables representing “Healthiness conditions” which must be always satisfied. Inconsistencies may arise in the formal specifications due to a contradiction in the state invariant. Other possible source of inconsistencies are the violation of the state invariants. These different kinds of inconsistencies are easy to check and correct using formal notations like Z. We shall provide an example to explain this situation.

Let's revisit the Link schema. This schema includes:

- Names of internal (inside the actor) elements int_components which are linked with the considered element
- Names of external (from SD model) dependencies ext_components which are linked with the considered element
- Type of the link
- Names of elements which give positive (contrib_p) and negative (contrib_n) contribution to the softgoals.

Link
ΦDepend
int_components, ext_components : PROP NAME
contrib_p, contrib_n : PROP NAME
link : LINK_TYPE

link = task_decomp ⇒ type = task
link = contrib ⇒ type = softgoal
contrib_p ∪ contrib_n ≠ ∅ ⇒ link = contrib ∧
⟨contrib_p, contrib_n⟩ partitions int_components
ext_components ≠ ∅ ⇒ type = task
link = NA ⇔ int_components = ∅
The predicate part describes the following constraints between types of links and types of elements:

- Task decomposition links are used only for tasks
- Positive or negative contribution is possible only for softgoals
- Tasks are used in connection with external components.

Now if an additional invariant is accidently introduced that states that “Positive or negative contributions are possible only for goals”, then due to this invariant, there will be contradiction in the invariants of the Link Schema, which will lead to inconsistencies in the specifications. These cases can be easily detected using formal methods. The other positive aspect is that when we use state invariants in the Z schema, we are making specifications precise and formal, which are not amenable to inconsistencies and it is easy to detect any inconsistencies in the specifications.

Existing tool support for Z, on the other hand, allows analysis of specifications without any additional effort. By making use of formal notation like Z to formalise the \( \ast \) diagrams, we are using the customary facilities available for Z like:

- type checking the components
- proving properties in relation to the components and
- providing precise rules for manipulating the components

For realising above-mentioned objectives, various tools for formatting, type-checking and aiding proofs in Z are available. We are listing some of them that might be used. First of them is CADiZ [77], which is a UNIX based tool for checking and typesetting Z specifications. Zola the WYSIWYG editor is another interesting tool, which supports the production and typesetting of Z specifications. Also included are a type-checker and a Tactical Proof System (available from http://www.ist.co.uk/PRODUCTS/zola.html). The integration of \( \ast \) diagrams and Z allows one to use Z type checkers like ZTC [74] and Z animation tools like ZANS [75] to analyse the models. It is projected to be compliant with the second edition of Spivey’s Z reference manual. Formaliser [53] is a syntax-directed Z editor as well as an interactive type-checker, running under Microsoft Windows obtainable from Logica. Z/EVES [106] supports the analysis of Z specifications in several ways: for syntax and type checking, schema expansion, and precondition calculation. Even with domain checking, many of the proof obligations
are easily proven. In more difficult cases, generating the proof obligation is often a substantial aid in determining whether a specification is meaningful.

Reasoning about the internal consistency of a formal specification written in Z is conducted primarily from the software developers point of view. We are aware that inconsistencies in the specifications may arise from an ill conceived problem specification resulting from the lack of understanding of the object of specification and incorrect formalisation.

5.4 Summary

This chapter introduced a methodology supporting the co-evolution of $i^*$ models and Z specifications. We also discussed how consistency is preserved during the co-evolution of formal and informal models.

In the next chapter, we shall give a proposal about the Combined use of Agent-Oriented Conceptual Modelling with the UML Sequence Diagram. This chapter explores how these two complementary approaches might be used in a synergistic fashion for requirements engineering.
Agent-oriented conceptual modelling (AOCM) offers an interesting approach to modelling early phase requirements. It is mainly effective in capturing organisational contexts, stakeholder intentions and rationale. In contrast, UML sequence diagrams are best suited for showing objects within the intended system interacting to perform the behaviour of a use case chronologically. In this chapter, we suggest an approach to translate sequence agnostic $i^*$ models into UML sequence diagrams using effect annotations. This allows us to benefit from the complementary representational capabilities of the two frameworks.

6.1 Introduction

As discussed earlier in the thesis, agent-oriented conceptual modelling notations are highly effective in representing requirements from an intentional stance and answering questions such as what goals exists, how key actors depend on each other and what alternatives must be considered [153]. These critical modelling decisions are taken in the early-phase requirements engineering. Most of the available modelling approaches are designed to work in the late phase of requirements engineering. As pointed earlier that the focus in the “late phase” is on the completeness, consistency and automated verification of the requirements [153]. Hence, it would be appropriate to present different modelling and reasoning support for the two phases. The SD model (see Figure 6.1) captures the social context of the system. As an example, consider a simplified version of the computer based training system for volunteers. This example will be used to illustrate both the $i^*$ notation and our proposed methodology for transforming $i^*$ models into UML sequence diagrams using effect annotations. An SR model (see Figure 6.2) provides a more detailed level of modelling by looking “inside” actors to model internal
intentional relationships.

A sequence diagram is a dynamic model that illustrates the interaction between objects arranged in a time sequence [40]. Sequence diagrams can be drawn at different levels of detail and to meet different purposes at several stages in the development life cycle [122]. A diverse set of stakeholders such as analysts, designers and users can easily understand UML sequence diagrams and are comfortable with them [40]. The documentation containing sequence diagrams is also extremely helpful when handing over a system to another person or organisation.

The Tropos methodology [23] explores how $i^*$ models might be refined to form the basis for late-phase requirements specifications and subsequently architectural specifications. The $i^*$ notation alone is not adequate for representing the level of detail necessary for late phase requirements specifications. To address this, formal languages such as Formal Tropos [55] have been developed. Alternative approaches have been proposed to define methodologies for transforming $i^*$ models into agent programs in formal agent programming languages such as ConGolog [145], AgentSpeak(L) [125] and 3APL [61]. The CREWS project [114, 120] and the CREWS-L’ Ecritoire approach discovers requirements specifications by allowing movements from goals to scenarios and vice-versa. These approaches do not make use of the intentional modelling concepts for arriving at the goals and scenarios for the system being developed, but are used to understand, model and validate the requirements provided by the user. Two approaches [49, 126] describing the derivation of use cases from the $i^*$ framework are also reported in the literature. Use case descriptions provide a specification of functional requirements in a form explainable and verifiable with the system stakeholders or sponsors. While UML use cases can describe scenarios which relate to goals/tasks, these descriptions are expressed in natural language. This introduces a number of fundamental problems. One is that there will be no effective way of specifying goals which may impact on more than one use case. The second is that the goal and use case relationship is not defined in terms of the organisational context in which the end application is going to be used. Thus, we would have no effective way of specifying the impact of the use case within a specific context. UML [122] does however provide a modelling technique that could potentially help in overcoming these drawbacks: the sequence diagrams. According to the viewpoint approach [48, 101, 102, 103], a viewpoint combines the idea of a view (or perspective) of a problem domain together with an owner for that view. Inconsistencies may arise in the overlaps among viewpoints, permitting local reasoning within each viewpoint. Inter-viewpoint consistency checking
is supported using a set of heuristic rules. We do not make such assumptions in our work. The viewpoint approach does not take into consideration the dynamic nature of the process whereas our work does.

Our thesis in this paper is that the sequence diagram notation in UML and the $i^*$ modelling framework can function in a complementary and synergistic fashion and the conceptual modelling methodology that supports their co-evolution is of interest [84]. As goal-orientation is completely missing in UML and the solution-oriented view, with the associated notions of messages and sequence is missing in $i^*$, both techniques can be considered as complementary to each other. A good integration would entail gains both for early-phase and late-phase requirements engineering and would allow a smooth transition from early to late requirements. The proposed approach makes use of the advantages of $i^*$ for the early phase of requirements engineering and then continues with the analysis and validation of requirements using sequence diagrams and scenarios. Scenarios are a set of hypothetical circumstances that require the use of the system in order to solve some problem. Appropriately aggregated sequence diagrams describing various realisations of scenarios from the $i^*$ diagram permit us to specify, understand, model and validate the requirements provided by the user in a more effective manner. We contend that a single notation is not rich enough to support the complete activities of requirements engineering, and that multi-notational modelling approaches are more suitable. When proposing the use of two otherwise disparate approaches for requirements engineering, we need to maintain consistency between the two approaches. A methodology is defined to help ensure consistency between the two models by constraining how an analyst might map the elements of the $i^*$ model to UML sequence diagrams. This allows us to understand how changes to a model in one notation might impact models in the other notation.

The proposed methodology uses the notion of cumulative effect annotation to determine whether the $i^*$ models and UML sequence diagrams are consistent with respect to each other. An effect is the result (outcome) of an activity being executed by some agent. It indicates the achievement of a certain environmental state communicated through an event. In our work, every goal/task/resource dependency must have an effect annotation. A causal relationship exists between an activity and an effect. An activity can cause many effects and an effect can be caused by a number of activities conjointly or disjointly. We introduce two notions of consistency in this thesis. The weak notion of consistency ensures that the $i^*$ model and corresponding sequence diagram do not make contradictory statements about the situations being modelled. The
6.2 Consistency Checking

A key element of any approach to deriving a model (of the same situation) in one notation from a model in a different notation is a clear articulation of what it means for models in these notations to be consistent. In the following, we define a notion of consistency between \( i^* \) models and sequence diagrams. In a loose sense, this notion of consistency provides a (partial) semantic account of correctness of the mapping process. We will show later that any sequence diagram obtained from an \( i^* \) model using our proposed methodology is guaranteed to be consistent (under this notion of consistency) with the \( i^* \) model it is obtained from.

Our definition of consistency relies on effect annotations of both \( i^* \) models and sequence diagrams. An Effect annotation is a formal or informal description of the effects of the execution/fulfillment/completion of certain elements of the proposed system and
6.2. Consistency Checking

Figure 6.2: SR model of the computer based training system

its environment. We are interested in three kinds of effect annotations. First, we annotate sequence diagrams with the effects of messages. These are informational effects - a message from one object to another has the effect of providing the latter object with some information (note that queries can also be viewed as information). Second, we annotate tasks in $i^*$ SR models with effects. Finally, we annotate dependencies in $i^*$ SD models with effects. Effects of dependencies are similar to Formal Tropos dependency fulfillment conditions - they describe what happens when a dependency is fulfilled. In this chapter, we only use informal annotation. Thus, when we refer to one set of effects entailing another (or being consistent with) we assume manual checking of entailment/consistency. If formal effect annotations are used, we would be able to use formal machinery for entailment/consistency checking - such as theorem proving.

In sequence diagrams, we annotate each message with its effects. We will refer to these annotations as cumulative effect annotations if they represent the accumulation of the effects of all previous messages on the same timeline (i.e. corresponding to a given object). Note that an effect annotation is made for each point in a timeline where a message is received.

An $i^*$ model ($M$) and a sequence diagram ($D$) are deemed to be consistent if the following are true:

Step I: There must exist an object in $D$ corresponding to the (unique) system actor
in $M$. We assume that we shall only deal with $i^*$ models with a unique system actor. Note that this does not lead to loss of generality.

Step II: For each actor that participates (either as depender or a dependee) in a dependency with the system actor in $M$, there must exist a corresponding object in $D$.

Step III: For each goal/task/resource dependency in $M$, there must exist a cumulative effect annotation $e$ in $D$ such that $e$, together with the effect annotation of the (always unique) task in the dependee actor to which this dependency is related to is consistent with the fulfillment condition of the dependency. This represents the \textit{weak} notion of consistency. In general the effects of a dependency may be achieved in multiple ways. They may be achieved by the effects of a task that is internal to an actor, more specifically the task within the dependee actor to which a dependency is directly related. Alternatively, the effects of this task together with the effects of messages between the actors involved in the dependency may achieve the effects of the dependency.

This notion of consistency is \textit{weak} in several ways. We only require consistency (either informally verified or relying on the semantics of the formal messages involved) between the effects of the dependency, the task and the messages involved. The $i^*$ model would usually reveal the task that is related to the dependency, but the messages involved (within the sequence diagram) in fulfilling the dependency are not immediately obvious. In the \textit{weak} notion of consistency, we only require that there exist a point in time in the sequence of message exchanges where the \textit{cumulative effect} is consistent with the task and dependency effects in question. It permits the possibility of the dependency \textit{effects} being realised by the combination of task and message \textit{effects}, but does not guarantee this to be the case. We define next an alternative, \textit{stronger} notion of consistency which actually offers such a guarantee.

Step $\text{III'}$: For each goal/task/resource dependency in $M$, where $e_M$ is the \textit{effect} of the (always unique) task in the dependee actor to which the dependency is related, at least one of the following must be true:

1. $e_M$ entails the \textit{effects} of the dependency, or

2. $e_M$ alone does not entail the \textit{effects} of the dependency, but $e_M$ together with an effect annotation $e_D$ in the timeline of the object corresponding to the \textit{dependee} actor and/or an effect annotation $e'_D$ in the timeline of the object corresponding to the \textit{dependee} actor entails the \textit{effects} of the dependency. This represents the \textit{strong} notion of consistency.
Table 6.1: Key steps in the proposed methodology of mapping $i^*$ model into UML sequence diagram

<table>
<thead>
<tr>
<th>Key steps in the proposed methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create objects corresponding to $i^*$ actors.</td>
</tr>
<tr>
<td>2. Identify key functionalities of the system actor in the $i^*$ model.</td>
</tr>
<tr>
<td>3. Identify all goal/task/resource dependencies between a system actor and any other actor.</td>
</tr>
<tr>
<td>4. Annotate dependencies with effects in the $i^*$ model.</td>
</tr>
<tr>
<td>5. Annotate tasks with effects in the $i^*$ model.</td>
</tr>
<tr>
<td>6. Annotate the sequence diagram with cumulative effect annotation for each message.</td>
</tr>
<tr>
<td>7. Perform the evaluation of alternatives based on softgoals.</td>
</tr>
<tr>
<td>8. Perform the evaluation of alternatives based on means end links.</td>
</tr>
<tr>
<td>9. Apply effect annotation to identify additional communication links for consistency analysis.</td>
</tr>
<tr>
<td>10. Validate sequence diagrams with the aid of walk through scenarios.</td>
</tr>
</tbody>
</table>

6.3 Proposed Methodology

In this section, we shall present a methodology for mapping $i^*$ models to sequence diagrams. In effect, the methodology defines how a sequence diagram might be generated in a manner constrained by the $i^*$ model on offer.

Step 1: Create objects corresponding to $i^*$ actors: We need to create a sequence diagram object for the system actor and for every actor that relates via a dependency with the system actor.

Step 2: Identify key functionalities of the system actor in the $i^*$ model: Within the system actor identified on the SR model, we select the highest-level task(s). In instances where there is more than one high level task, we include those tasks as well. This is because collectively they indicate the key functionalities that the system will
offer. All tasks that are further refined using decomposition links from those high level task(s) is additional system functionalities.

Step 3: Identify all goal/task/resource dependencies between a system actor and any other actor: We now identify all goal/task/resource dependencies between the system and any actor that interacts with the system. Tasks usually lead to exchange of messages between the system actor and other actors. In the case where the system is the dependee the message link becomes a service request on the system. These service requests may be implemented by an interface to the user or they may be implemented as web services. In the case where the system is the depender, the task dependency will be a method invocation on the external actor. Such a method invocation could be a notification sent to a human user via SMS or a notification sent to a system actor via a web service call.

Resource dependencies that exist between the system and actors identify information resources that either must be provided to the system by another actor, or that a system provides to another actor. In the case where the system is the dependee, the system will have to provide information to an actor. In the case where the system is the depender, the system will have to accept information from another actor. Currently we need to make a note of such system requirements. Using the UML sequence diagram we can show data being sent out as an internal method call within the system (since we have not yet decided on how to send the data out). As in the case of task dependencies, the SR diagram will provide sufficient information as to the purpose of the information resource being provided.

Step 4: Annotate dependencies with effects in the i* model.
Step 5: Annotate tasks with effects in the i* model.
Step 6: Annotate the sequence diagram with cumulative effect annotation for each message.

Step 7: Perform the evaluation of alternatives based on softgoal: Softgoals in much the same way cannot be mapped directly into UML sequence diagrams. We view softgoals as optimisation goals where there is no way of actually specifying whether the softgoal was achieved completely. Assume a case where a task contributes positively to the achievement of a softgoal for a system. Then it is desirable that the system design implements that task because we would be contributing positively to the achievement of a softgoal. In a case where there are multiple tasks contributing positively to the achievement of a softgoal we are faced with a design decision. Since each such task will contribute positively to the goal we need to evaluate which of the tasks should be
A task, which contributes negatively to a softgoal, should be avoided in the system implementation and further design and elaboration work should ensure that the task is not implemented. There may however be scenarios where a task needs to be implemented, as a result of another dependency which will contribute negatively to a softgoal. At that stage domain experts and stakeholders will need to outweigh the advantages against the disadvantages of not achieving the specific softgoal.

We do not propose a new way of dealing with softgoals during requirements engineering. The ideas in this procedure are complementary to other approaches which specifically focus on softgoals during requirements engineering. One such approach is the NFR framework [26, 27, 28].

Step 8: Perform the evaluation of alternatives based on means end links: Multiple means end links present alternative designs. Assume a goal G which can be satisfied through one of the possible two tasks (task H and task I) through two means end links. This in turn means that a system could be designed implementing task H or it could be designed implementing task I. During the initial mapping into the UML sequence diagrams we could pick either of the two tasks, however domain experts could evaluate this decision later on and an alternative design may be adopted. At the requirements level it is sufficient for us to capture one task, preferably one which reuses other tasks throughout the system or contributes to the overall design of the system. However we still present the development team with a choice for picking the design which is suited more closely with their goals.

Step 9: Apply effect annotation to identify additional communication links for consistency analysis: Derive sequence diagram messages using effect annotations and dependencies. We generate the messages in the sequence diagram in two phases. First, we analyse goal/task/resource dependencies identified in the process steps bearing in mind the choices made while evaluating alternatives. These dependencies can suggest obvious messages between the objects corresponding to the depender and dependee actors. Second, we consider the effect annotations of such dependencies that have been identified as being of interest in the previous steps. If these effects are achieved by the effects of the related (internal) task in the dependee actor, then no messages need to be added. Otherwise, we identify those (informational) effects that, together with the internal task effects, can lead to the achievement of the effects of the dependency. These informational effects immediately suggest the nature of the message that would achieve such effects. We provide a detailed example of this methodology in the next
6.4 Case Study Describing our Approach

Consider a simplified version of the computer based training system (CBT system) for volunteers. The SR model of the system is shown in Figure 6.2. The SR model includes three actors, the TrainingCoordinator, the TrainingSystem and the Volunteer. The SR model also identifies 12 dependencies, however for the purpose of our methodology we are only concerned with the 11 dependencies that involve the system actor. These dependencies are: OnlineTrainingConducted, TrainingContent, TrainingScheduled, TrainingAttended, ConductTraining, TrainingContentEasyToUse, TrainingScheduleReminder, Confirmation, Username&Password, TrainingLesson, TrainingInformation. We shall now apply the procedures in the outlined methodology to create UML sequence diagrams from the computer based training system i* model.

Step 1: We create an object for the system actor named TrainingSystem, an object for the TrainingCoordinator and an object for the Volunteer actor.

Step 2: Looking at the SR diagram in Figure 6.2 we identify the main task as being OrganiseTraining with additional system tasks being ImpartTraining, MaintainSchedule, ObtainConfirmation and Create & ForwardUserAccessInformation.

Step 3: From the SR model we can see that the only task dependency that exists is the ConductTraining dependency between the Volunteer and the TrainingSystem. In the dependency relationship the system is the dependee and as such the task dependency becomes a service call on the system. The resource dependencies we have identified are, TrainingContent, Confirmation, TrainingScheduleReminder, Username&Password, TrainingLesson and TrainingInformation. For the dependencies in which the system is the dependee the system will need to implement a way of displaying the resource information to the relevant actor. Dependencies in which the system is the
6.4. Case Study Describing our Approach

Table 6.2: Effect annotation of dependencies

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnlineTrainingConducted</td>
<td>The training has been conducted by the training system.</td>
</tr>
<tr>
<td>TrainingContent</td>
<td>The training content has been provided by the TrainingCoordinator to the training system.</td>
</tr>
<tr>
<td>TrainingScheduled</td>
<td>The training has been scheduled by the training system.</td>
</tr>
<tr>
<td>TrainingAttended</td>
<td>Volunteers have attended the training.</td>
</tr>
<tr>
<td>ConductTraining</td>
<td>The training has been conducted by the training system.</td>
</tr>
<tr>
<td>TrainingScheduleReminder</td>
<td>The reminder for scheduled training sessions has been sent to participating volunteers by the training system.</td>
</tr>
<tr>
<td>Confirmation</td>
<td>The volunteer has provided a confirmation to the system for attending a training session.</td>
</tr>
<tr>
<td>Username&amp;Password</td>
<td>The system provides the user with their account details.</td>
</tr>
<tr>
<td>TrainingLesson</td>
<td>The training lesson has been provided to the volunteer by the training system.</td>
</tr>
<tr>
<td>TrainingInformation</td>
<td>The training information has been provided to the volunteer by the training system.</td>
</tr>
</tbody>
</table>

For the dependencies in which the system is the depender the system will need to implement a way of invoking a method on the external actor. Dependencies in which the system is the depender are, the TrainingContent dependency and the Confirmation dependency.

Step 4: We are using natural language to annotate effects of dependencies in the $i^*$ model. We capture the dependencies and effects in a table format as shown in Table 6.2 above. Since TrainingContentEasyToUse is a softgoal dependency we do not include it here because it cannot be annotated with an effect. This is due to the fact that there is no way to specify the successful achievement of a softgoal.

Step 5: We also use natural language to annotate effects on tasks in the $i^*$ model (SR model). For each dependency identified in the previous step we annotate the internal task in the dependee with effects. The internal tasks and annotations are outlined in Table 6.3.
### Table 6.3: Effect annotations of tasks (internal tasks)

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Task</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnlineTraining Conducted</td>
<td>ImpartTraining</td>
<td>The training coordinator has provided the training content and the training lessons have been provided by the system to the volunteer.</td>
</tr>
<tr>
<td>TrainingContent</td>
<td>GenerateTraining Content</td>
<td>The training content is generated by the training coordinator and provided to the training system.</td>
</tr>
<tr>
<td>TrainingScheduled</td>
<td>MaintainSchedule</td>
<td>The training schedule reminders have been sent out to the volunteers.</td>
</tr>
<tr>
<td>TrainingAttended</td>
<td>AcquireTrainingSkills</td>
<td>The volunteer is ready for participating in a training lesson and provides confirmation that he will be attending a scheduled training session.</td>
</tr>
<tr>
<td>ConductTraining</td>
<td>OrganiseTraining</td>
<td>Training schedule is maintained, confirmation about participating volunteers is received, user accounts for participating volunteers are created and the training is imparted.</td>
</tr>
<tr>
<td>TrainingScheduleReminder</td>
<td>MaintainSchedule</td>
<td>The schedule for training lessons is maintained to ensure no double bookings occur.</td>
</tr>
<tr>
<td>Confirmation</td>
<td>ProvideConfirmation</td>
<td>Participating volunteer provides confirmation to the system that he/she will be attending a scheduled training session.</td>
</tr>
<tr>
<td>Username &amp; Password</td>
<td>Create &amp; ForwardUserAccessInformation</td>
<td>The system creates an account for participating volunteer.</td>
</tr>
<tr>
<td>TrainingLesson</td>
<td>ImpartTraining</td>
<td>The training is provided to the volunteer by the system.</td>
</tr>
<tr>
<td>TrainingInformation</td>
<td>OrganiseTraining</td>
<td>The training information is provided to the user once the schedule has been organised, participants have been registered and training has been imparted.</td>
</tr>
</tbody>
</table>
6.4. Case Study Describing our Approach

Step 6: For messages that relate to goal/task/resource dependencies, we simply place the dependencies effect as the effect on the messages. For instance the TrainingContent resource dependency will have a message in the UML sequence diagram named trainingContent(). The effect of the resource dependency will be used to provide the effect of the message. For messages which represent internal tasks in the system actor, we will need to create new effect annotations. For instance the task Create&ForwardUserAccessInfo in the training system actor will have a method on the UML sequence diagram as Create&ForwardUserAccessInfo(). The effect of this message is created by the analyst while performing this step. An example effect could be that the successful completion of the message creates an account for the participating volunteer in the system.

Step 7: The SR model for the training system does not present any softgoals of concern to the development of the system. However for this example we assume a very simple scenario where a softgoal is positively contributed to by a task A and negatively contributed to by a task B. Effectively our design would aim to implement task A. However task A may be contributing negatively to another softgoal of greater importance and as such it is not easy to determine which task should be implemented.

To resolve these problems, accepted frameworks such as the NFR framework [26, 27, 28] should be used. They present a proven and well-documented way of dealing with softgoals during requirements engineering.

Step 8: As with softgoals the training system has no means end links. We again refer to an example to explain this procedure. Assume task A which can be completed by either task B or task C. In order to determine which task is to be implemented we draw on the expertise of domain experts. Task A for instance may require additional components to be developed for the software system, while task B in contrast may be reusing existing components. In such a situation it would be more appropriate to implement task B. Effectively these means end links present a design decision which can be addressed presently at a high level but may be elaborated at a later stage of development.

Step 9: We shall now use the effect annotations on the i* model and the UML sequence diagram to determine additional information effect messages. This analysis is helpful towards the evaluation of consistency preservation in our methodology. Consider the dependency OnlineTrainingConducted with the effect being as follows; The scheduled training has been conducted by the training system. The task in the dependee to which this dependency is linked is the ImpartTraining task in the training
system actor. The effect of the *ImpartTraining* task is as follows; The training content has been provided by the training coordinator and the training lessons have been provided by the system to the volunteer. The effect of the *OnlineTrainingConducted* dependency implies that the *TrainingCoordinator* who is the depender, will be notified of the training be scheduled. Otherwise the depender has no way of knowing that the effect of the dependency has been achieved. As a result we introduce another message to be added to our sequence diagram. The message is named notifyTrainingCoordinator(). The effect of this message is that the successful completion of a scheduled training session is communicated to the *TrainingCoordinator*. The effect of this message together with the effect of the *ImpartTraining* task can now achieve the effect of the *TrainingConducted* dependency. This informational effect message guides to the notion of consistency preservation in our proposed methodology. An initial sequence diagram for the *TrainingSystem* is shown in Figure 6.3.

We shall use this sequence diagram and annotate it with effects in order to identify which goals can be satisfied by the system. The annotated sequence diagram for the *TrainingSystem* is shown in Figure 6.4. It is annotated with information acquired in step 6 of our methodology.

Step 10: We now proceed to validate our sequence diagram using a walk through
6.4. Case Study Describing our Approach

Figure 6.4: Annotated computer based training system sequence diagram

scenario. We assume a scenario where a Volunteer accesses the training system and invokes the conductTraining() method. The system will then prompt for the volunteer’s confirmation to ensure the right request has been received. The volunteer can use the provideConfirmation() method to notify the system to proceed with the OrganiseTraining task. The system will then create a user account for the volunteer by prompting the volunteer for his details, including the username and password. The volunteer is able to provide his information to the system using the username&Password() method. Once all the necessary information is provided the system may send out a reminder using the trainingScheduleReminder() method in circumstances where the training was not taken at the time the volunteer registered with the system. The maintainSchedule() task is then performed by the system to ensure the schedule of training sessions is kept up to date. At some time before the training the TrainingCoordinator is able to load the training information to the system using the trainingContent() method. The system will then start to notify all volunteer’s registered for the training session via the impartTraining() method. The trainingLesson() method is then invoked and the lesson is started and the trainingInformation() method is used to provide the actual training information to an interface where the users can view it. The scenario concludes when the TrainingCoordinator is notified of the successful completion of the scheduled
training lesson with the notifyTrainingCoordinator() method.

The benefits of the synergy discussed in this chapter are evident on numerous fronts. During requirements engineering the efforts provide a way of validating requirements. This is done by walking stakeholders through the functionality of the proposed system that will be available once the system is designed. During the analysis and design stages the creation of high-level technical models will provide the system architects a starting point for elaborating system functionality and system architecture. Moreover, the starting point will be consistent with the requirements identified.

A scenario-based approach is well suited for a test driven development environment. Test cases can be captured early on to ensure final system is meeting the identified requirements.

We have tried to provide methodology, which is applicable over a broad spectrum of problem domains. We maintain however that a high degree of human involvement is needed to ensure that the mapping from i* models into UML sequence diagrams is significant. This is because human actors could provide ongoing problem solving support in case of emerging conflicts and inconsistencies. This process may be made easier by providing a tool support for these activities.

6.5 Summary

We have presented an approach using two complementary notations, one used for modelling early phase requirements and the other for the analysis of requirements. The i* model is well suited for capturing the strategic intent of the stakeholders. It is also able to provide a basis for further architectural work on the system to be designed. UML sequence diagrams in contrast are well suited for realising requirement scenarios. They are also useful in elaborating higher-level functional requirements. Combining the two approaches provides an effective methodology for evaluating the requirements captured in the i* model. A set of sequence diagrams describing various scenarios of the i* diagram permit us to specify, understand, model and validate the requirements provided by the user in a more effective manner.
Chapter 7

Conclusions and Future Work

7.1 Contributions

In this thesis, we focused on devising novel elicitation techniques for the agent-oriented conceptual modelling language. We also focused on the methodologies that permit models in distinct notations (ideally with complementary representational capabilities) to co-evolve. This research uses the notion of co-evolution in a very specific sense to describe a class of methodologies that permit $i^*$ modelling to proceed independently of specification in a distinct notation, while maintaining some modicum of loose coupling via consistency constraints. In the current instance, this research examines how this might be done with formal specification notations (specifically Z) as well as an industry standard modelling language (UML). The major contributions of this work are listed below:

- In this thesis, we have focused on presenting both arguments for the use of agent-oriented conceptual modelling and proposals for devising robust early-phase requirements elicitation methodologies. We have argued for the use of elicitation templates that reflect the information requirements of the underlying modelling notation to ease the elicitation process. We have also provided a notion of an organisational model that assists modellers and stakeholders during the requirements gathering process (interview process). We have also argued for the use of enterprise ontologies to drive the elicitation process. These proposals can be generalised to other conceptual modelling contexts (using distinct modelling notations to the one being used in this proposal).

- We propose a methodology for the combined use of agent-oriented conceptual modelling and the Z notation in requirements engineering. The Z notation permits us to specify requirements with a degree of precision and formality that $i^*$ does not. The mapping process that we have described so far leads to a Z
7.2 Requirements Validation

We suggest a hybrid modelling, or co-evolution, approach in which $i^*$ models and Z specifications (as well as UML sequence diagrams) are concurrently maintained and updated, while retaining some modicum of loose consistency between the two. This allows us to benefit from the complementary representational capabilities of the two frameworks. We are also interested in *compositional, extensible* and easily *maintainable* modelling frameworks. We claim that the combination of high-level modelling in $i^*$ coupled with specifications of functionality using Z specifications (as well as UML sequence diagrams) offers such a framework. This notion of co-evolution of models contributes to requirements validation as well. One perspective on requirements validation is to assess the degree to which requirements goals satisfy functional and non-functional
requirements (NFR), as demonstrated in the \( i^* \) models where the association between functional requirements and softgoals (i.e. NFRs) can be established by inspecting strategic dependency and strategic rationale models that show goals, actors/agents, tasks and dependency relationships. The \( i^* \) framework provides some modicum of support for assessing dependencies, but most validation still requires human expertise. Sommerville has identified some requirements validation checks in [131] to be performed on requirements. Verifiability checks reduce the potential for dispute between the stakeholder and analyst. Verifiability checks can be easily carried out by using the novel approach to requirements elicitation proposed in this thesis (chapter 2). Verifiability checks can also be carried out by using the proposed approach in chapter 6, which makes use of the advantages of \( i^* \) for the early phase of requirements engineering and then continues with the analysis and validation of requirements using sequence diagrams and scenarios. A key element of any approach to deriving a model (of the same situation) in one notation from a model in a different notation is a clear articulation of what it means for models in these notations to be consistent. In this work, we have clearly defined the notion of consistency between the \( i^* \) models and Z specifications (as well as UML sequence diagrams). Consistency checks can be easily carried out using the approach proposed in chapters 5 & 6 (as identified by Sommerville in [131]).

### 7.3 Combined use of Proposed Methods

This thesis illustrates a set of methods that can assist requirements engineering. This work proves that different methodologies can be used at different stages of requirements engineering, thus exploiting the strengths of each methodology. The problem of how to deal with changing requirements is also well illustrated through ‘co-evolution’ strategies that allow translations from one method to another and back, maintaining integrity in requirements specifications over time. Figure 7.1 shows the schematic representation of various methods proposed in conjunction with each other:

1. **Elicit requirements using Requirement Capture Templates**: We have presented a novel approach to requirements elicitation in an agent-oriented conceptual modelling context based on the specific outcomes of our experience with early-phase requirements engineering in the emergency services domain. We show how it is possible to use the underlying modelling notation to drive the design
of a set of Requirements Capture Templates that can simplify the process of requirements elicitation via stakeholder interaction. In effect, these templates are forms that an analyst/modeller seeks to complete in the course of a stakeholder interview which provide structure and direction to an otherwise unstructured exercise.

2. Develop Strategic Dependency (SD) & Strategic Rationale (SR) models: Once completed, these templates can be (manually) transformed in a relatively straightforward fashion to eventual SD and SR models (i* model). In addition, these completed templates can serve as a structured repository and record of stakeholder interaction that can be revisited whenever requirements need to be re-negotiated or revised (for instance, to resolve an inconsistency) or business processes need to be re-engineered. We show that these templates can themselves be derived from enterprise/domain ontologies and organisational models, and interact with these in interesting ways.

3. Combined use of i* model and Z specification (as well as UML sequence diagram): The i* model obtained from the previous activities can be used as starting point for this stage. We have proposed two methodologies which can be used in parallel to each other (as can be seen in Figure ). We explore how the i* framework might complement rather than supplant existing, industry-standard requirements modelling and specification techniques. In so doing, we deviate somewhat from the approach taken by projects such as Tropos [23, 24], which
uses the $i^*$ notation to represent early- and late-phase requirements, architectures and detailed designs. We believe that the value of conceptual modelling in the $i^*$ framework lies in its use as a notation complementary to existing specification languages, i.e., the expressive power of $i^*$ complements that of existing notations. The use of $i^*$ in this fashion requires that we define methodologies that support the co-evolution of $i^*$ models with more traditional specification and modelling languages. We use the notion of co-evolution in a very specific sense to describe a class of methodologies that permit $i^*$ modelling to proceed independently of specification in a distinct notation, while maintaining some modicum of loose coupling between them. We attempt to validate this approach by exploring co-evolution of $i^*$ models with models in two very different modelling/specification languages. At the formal end of the spectrum, we explore the co-evolution of $i^*$ models with formal specifications in Z [132]. Our aim, is to support the modelling of organisational contexts, intentions and rationale in $i^*$, while traditional specifications of functionality and design proceeds in the formal notation. We note that many of the lessons learnt from this exercise apply to other formal methods as well. At the other end of the spectrum, we explore the co-evolution of $i^*$ models with models in the industry-standard (and largely semi-formal or informal) UML notation. We specifically focus on UML sequence diagrams, where the representational capabilities are clearly complementary to those of $i^*$ ($i^*$ models are sequence-agnostic).

4. **Consistency checking:** A key element of any approach to deriving a model (of the same situation) in one notation from a model in a different notation is a clear articulation of what it means for models in these notations to be consistent. We define a notion of consistency between $i^*$ models and Z specification (as well as UML sequence diagrams). In a loose sense, this notion of consistency provides a (partial) semantic account of correctness of the mapping process. Any sequence diagram (or Z specification) obtained from an $i^*$ model using the proposed methodology is guaranteed to be consistent (under this notion of consistency) with the $i^*$ model it is obtained from.

The methodologies outlined in this thesis are capable of guiding the early phase requirements process to a point where solution designs can be identified and discussed with relevant stakeholders. They are particularly good at ensuring that solution designs are achieving the required effects, specified on the $i^*$ model. This is aimed at identifying or dealing with of any major problems or risks that may impede the remaining section
7.4 Some Lessons Learned

Managing flood rescue and evacuation operation can be considered as a rich and challenging example for requirements engineering. As compared to conventional examples used in various research papers, it offers extraordinary combination of features found in real-life requirements engineering exercise. For example, the performance of the automated part (Call Taking Supervisor/System) of the combined system depends heavily on the satisfactory actions taken by the agents/actors in the environment. This case study presents some methodological principles and lessons derived from a real-life conceptual modelling exercise:

- **Incremental creation of knowledge and model:** The requirements gathering exercise began by conducting stakeholder interviews and discussions on the existing activities, systems and rationales. We agree that most of the RE activities are informal in nature. Modelling exercise benefited a lot as a result of observing practicing stakeholders and additional reading was undertaken to clarify these observations. We had constant interactions with the stakeholders during the requirements gathering as well as modelling exercise. Feedback from the stakeholders led to the development of conceptual model, which was closer to the users view. As a consequence the model was modified three times, reflecting upon the incremental creation of knowledge and model.

- **Observation:** We believe that the modellers or systems analysts bring theoretical and technical knowledge to the RE process. The stakeholders bring comprehensive knowledge of their domain and experience of the problem situation. Neither has superior knowledge, rather they are both proficient in their own domains; it is the interaction between these two group of people that forms the basis for the problem solving required from the RE process. We feel that it is in this area that further research will bring advance understanding of the RE process.

- **Hierarchically structured set of models:** A key element of the process of agent-oriented conceptual modelling of a large organisation is the development of a hierarchically structured set of models. In our instance, we started with a highest-level SD model, which treated the emergency services organisation as
a single agent that interacted with a variety of external/entities and agents. A corresponding SR model was built which detailed the goals, tasks, resources and softgoals internal to the emergency services actor and their relationships to external actors and external dependencies. This SR model did not expand on the internal characteristics of the other actors in any great detail. At the next level of abstraction, we constructed an SD model of the emergency services organisation where each individual department was modelled as a distinct actor. The requirements capture templates (RCTs) became useful from this point onwards. Implicit in the Requirements Capture Templates presented here is the notion of an organisational model. Several concepts explicitly referred to in the RCTs, such as departments/units, functions and activities derived from such a model are extraneous to the $i*$ notation. These notions make the RCTs conceptually accessible to organisational stakeholders, yet at a lower level lead to notions directly supported by $i*$. The underlying organisational model helps make the elicitation process more systematic. Building such a model explicitly is helpful, but not essential (i.e., the underlying organisational model can remain implicit in the RCTs). The agents responsible for the activities may be internal or external to the organisation.

- **Knowledge about the target actor:** It is important that the modeller or analyst should only elicit the relationships/dependencies the source actor (stakeholder being interviewed) has on the target actor(s). Our empirical evidence suggests that the source actor/agent may not have sufficient knowledge or be aware of the relationships/dependencies the target actor(s) has on the source actor/agent. Hence, to complete the relationship/dependency between the remaining actors in the given activity we use the same requirements capture form and conduct a similar requirements gathering process with the remaining actors/agents. Based on the same observation, it is important that the modeller should only elicit the intentional characteristics that are 'internal’ to the actor/agent (stakeholder being interviewed). Our empirical evidence suggests that the source actor/agent may not have sufficient knowledge of the intentional characteristics that are 'internal’ to other actors/agent in the given activity. Hence, to complete the intentional characteristics that are 'internal’ for the remaining actors/agents in the given activity it is proposed to use the same requirements capture form and conduct a similar requirements gathering process with the remaining actors/agent.
Structured repository: We have argued that the templates presented in this thesis can ease the requirements elicitation process. However, these templates serve other useful functions as well. They can provide a structured repository and record of stakeholder interviews that can be revisited when requirements must re-negotiated or revised (for instance, when changes are made to models, or when inconsistencies are detected). The detailed rationale recorded in these templates can also be of value in business process re-engineering. To anticipate and support future business process re-engineering efforts in the context of the emergency services agency, we are also detailing alternative solution scenarios by completing additional RCTs that answer “how-else” questions (while the primary RCTs represent the “as-is” scenarios).

Scalability issue of the models: In the current state, $i^*$ models might be difficult to analyse in the case of large enterprises, since there are too many modelling elements present in a model. These models can grow up quickly, complicating their analysis. This is reflected in the proposed methodologies as well.

Encapsulation: Encapsulation is the ability of the modelling technique for providing mechanism to use abstract concepts that represent a set of more concrete concepts. In the current state, $i^*$ models don’t have mechanisms in place for encapsulating modelling primitives. This leads to difficulty in understanding and evaluating the fragments of the model that represent various processes of the enterprise. This is reflected in the proposed methodologies as well.

Use of Standard specification languages/notations: The methodologies presented in this work make use of standard specification languages/notations (Z and UML). This provides common vocabulary for the analysts, designers and programmers to communicate. Without such a commonly/widely used reference specification languages, the communication and teamwork would have been much more difficult and time consuming.

7.5 Future Directions

Future work is necessary to fully recognise the benefits of the methodologies presented in this thesis. This involves work in the following areas:

- The issue of requirements evolution, discussion about how the structured repository and record of stakeholder interaction could be revisited when changes are
requested presents interesting future research challenges.

- With respect to the ontology co-evolution, the thesis presents with some interesting future research questions. For example, are currently existing ontologies enough? How does this fit with the current semantic web initiative, ontology-based approaches? In the future, we will work towards automating the approach proposed in this thesis.

- The work on co-evolution of formal specifications (specifically Z) and agent-oriented conceptual modelling have not investigated the possibility of articulating semantic consistency constraints between \( i^* \) models (possibly augmented with Formal Tropos annotations) and formal specifications. This presents interesting future research investigation.

- In the last chapter, we presented a methodology about the combined use of \( i^* \) with the UML sequence diagram using effect annotations. The complete incorporation of the notion of softgoals requires challenging future research investigation in the proposed integration process (between \( i^* \) and UML sequence diagram). The combined use of these two approaches could be made easier with appropriate tool support. Consistency checking using effect annotation is a prime candidate for automation in support tools. Future work will be directly aimed at providing automated support for the proposed methodology.

- In the work focussing on the combined use of \( i^* \) and Z specifications (as well as UML sequence diagram), realising provably sound and complete mapping functions and formally grounded machinery to support co-evolution represents a major research agenda. We present preliminary progress in that direction in this work.

- The methodologies presented should be extended to consider the other phases of software development (architectural and detailed design etc).
Appendix A

Requirements Capture Templates (RCT)

A.1 Objective

This document captures needs and requirements of a generic organisation.

A.2 Instructions for Organisational Unit Template

1. Complete the department information in the table supplied below.

2. Provide the department name, name of the department head and his/her designation, department rationales and list functions of the department.

A.3 Instructions for Function Elaboration Template

1. Provide the function name and describe its rationales (use separate sheet for each function).

2. List down all the activities taking place under each function.

3. Take one activity (from the selected function) at a time and give its brief description (use separate sheet for each activity).

4. Provide all the actors and agents involved in the selected activity. A list of unique actors is identified. Write down the rationales of the selected activity.

5. List down all the actors (Source and Target) involved, their relationships/dependencies (from source onto target actor) to achieve/satisfy the desired activity.

6. Provide any further additional information deemed necessary to give insight into the relationships (eg, importance of the relationship/dependency etc., for the success of the activity).
7. Repeat steps 4 to 6 for each the activities for the given function (by taking one activity at a time).

A.4 Instructions for Activity Elaboration Template

1. Provide all the actors and agents involved in the selected activity. A unique list of actors is identified (use separate sheet for each activity in a function).

2. Take one actor at a time involved in an activity; describe all the internal tasks/means by the actor to achieve the activity (use multiple rows to describe multiple internal tasks for the selected actor).

3. Write any other additional information on further task decomposition or means deemed necessary to give more insight.

4. Repeat step 1 to 3 for all the actors involved in the activity.

A.5 Definitions

1. Rationales: Rationales are the reasons, purpose and motivations behind carrying out particular function or an activity. Rationales are necessary to capture the underlying motivations and intents behind the evident activities and flows in the process.

2. Actor/Agent: Human being, hardware, software and other related objects in the environment the system functions in. The term Agent/Actor is used interchangeably to identify object in the environment.

3. Source Actor: The source actor assigns or depends on other actor(s) [called Target Actor(s)] to achieve or realise its goals or tasks.

4. Target Actor: The target actor is responsible to execute the task or goal assigned by the source actor.
Table A.1: Organisational Unit Template for Department Details

<table>
<thead>
<tr>
<th>Organisational Unit Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department Details</td>
</tr>
<tr>
<td>Department Name</td>
</tr>
<tr>
<td>Name of the Department head</td>
</tr>
<tr>
<td>Designation of Department head</td>
</tr>
<tr>
<td>Department Rationale</td>
</tr>
<tr>
<td>High-Level Functions of the Department</td>
</tr>
<tr>
<td>Modeller Signature</td>
</tr>
<tr>
<td>Stakeholder Signature</td>
</tr>
<tr>
<td>Department Name</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Function Name (Use separate sheet for each function)</td>
</tr>
<tr>
<td>Function Rationales (Use separate sheet for each function)</td>
</tr>
<tr>
<td>Activity Name and Description (Use separate sheet for each activity under the function)</td>
</tr>
</tbody>
</table>

Table A.2: Function Elaboration Template for Function Elaboration
### Table A.2: Function Elaboration Template for Function Elaboration (contd)

<table>
<thead>
<tr>
<th>Activity Rationales</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsible Actor(s) involved in the activity (Unique list of Actor(s))</td>
<td></td>
</tr>
</tbody>
</table>

**Relationship/dependencies between responsible actor(s) to achieve/satisfy the above activity**

(relationship is described as the dependency from source actor on to target actor, use separate row for each relationship and dependency)

<table>
<thead>
<tr>
<th>Source Actor</th>
<th>Relationship / Dependency</th>
<th>Target Actor</th>
<th>Additional information / elaboration on the relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Modeller Signature**

**Stakeholder Signature**
### Activity Elaboration Template

**Internal Intentional Characteristics of individual actor(s) to achieve the activity**

(Use multiple rows to describe multiple internal tasks for each actor)

<table>
<thead>
<tr>
<th>Department Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Name</td>
<td></td>
</tr>
<tr>
<td>Activity Name</td>
<td></td>
</tr>
</tbody>
</table>

**Responsible Actor(s) involved in the activity**

(Unique list of Actor(s))

<table>
<thead>
<tr>
<th>Actor</th>
<th>Internal Task/Means to achieve the activity by individual Actor</th>
<th>Additional information on task or means to achieve the activity or Actor Rationales</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Modeller Signature**

**Stakeholder Signature**
Appendix B

Z Notation

This introductory description of Z notation is based on [13, 20, 133, 134], which the reader may refer to for additional details. Z has been developed at Oxford University since the late 1970’s by members of the Programming Research Group (PRG) within the Computing Laboratory [62, 111, 133, 134]. A computer-based archive server at the Programming Research Group in Oxford is also available at the following 'URL' http://www.zuser.org/z/ for complete reference. Z is a typed language based on Zermelo-Fraenkel (ZF) set theory and first order predicate logic. The notation is useful to organise the thoughts and assist the communication of ideas within a design team. It is also readable enough to be used as a documentation tool in a manual. Natural language should also be included to provide an informal description of the system and to relate the mathematical descriptions to the real world.

The idea behind an abstract Z specification is to describe what a system does rather than how it does it. Imperative programming languages are specifications, but these concentrate on how the result is to be achieved. Functional programming languages are more like specification languages since these describe what result is required. However they are designed to be executable. Z can be used in a functional style. However it is possible (and sometimes desirable) to write non-deterministic specifications in Z. This means the exact execution of the specification cannot be determined. The Z notation is designed to be expressive and understandable (by humans) rather than to be executable. Even though a Z specification is not in general executable, by passing it around members of a design team it may be mentally executed and checked far more reliably than an equivalent informal specification [13].

Z includes a schema notation to aid the structuring of specifications. It has been found convenient to use a state or model based approach in representing Z schemas [13]. A system may be considered to be modelled as an abstract state and a sequence of operations on this state. In other words, schemas are primarily used to specify state spaces and operations for the mathematical modelling of the systems [13]. For example,
schema can be represented as shown below:

\[
\begin{array}{c}
\text{SchemaName} \\
x : T \\
y : Y \\
\hline
\text{Predicate}
\end{array}
\]

Hence, a Z schema consists of a schema name, declaration of variables, and a predicate. Predicates are used to state truth properties of values in a specification [20].

Subsequent discussion is based on [20].

The declaration \( x : T \) shows that \( x \) is of type \( T \), where \( T \) is a set. This is like saying \( x \in T \). A new basic type \( T \) is introduced to a specification by putting its name in square brackets:

\[ [T] \]

This allows us to name the types of a specification without saying what kind of objects they contain. For example, a specification of an address book might introduce the basic types Name and Address without worrying about the structure of these types:

\[ [Name, Address] \]

If we know the exact values of a type we use an enumerated type declaration:

\[
\text{Direction ::= north | south | west | east}
\]

Sets have types too. The type of the set \( \{1, 4, 5, 8\} \) is “set of N”- natural numbers. More precisely, this can be written as:

\[ \{1, 4, 5, 8\} : \mathbb{P}N \]

Types are used to structure specifications in Z [13]. Take an example of an english statement:

\( x \) is a path of least cost from A to B

we could give \( x \) a type:

\( x : Path \)

and constrain it further with a predicate:

\( \text{least}_\text{cost} x \)

In Z specifications, predicates are used to state truth properties of values in a specification. Examples of simple predicates may include:

- \( (z+2) = 7, z \in T \)
Compound predicates can be formed using the following logical operators:

- And $M \land N$ (Conjunction)
- Or $M \lor N$ (Disjunction)
- Implication $M \rightarrow N$

Logical-and is also known as conjunction and logical-or is known as disjunction.

Universal Quantification is written as follows:
\[
\forall X : T \bullet A
\]

This statement is true when $A$ holds for all the values $x$ of type $T$. In this case, $x$ is understood to be a quantified or bound variable.

Existential Quantification is written as follows:
\[
\exists X : T \bullet A
\]

This statement is true when $A$ holds for some value $x$ of type $T$.

Let's take some examples of Z specifications. The state variables of a counter system may be specified using the following schema [20].

\[
\begin{align*}
\text{Counter} \\
\text{ctr} : N \\
0 \leq \text{ctr} \leq \text{max}
\end{align*}
\]

Here, $\text{ctr}$ is declared to be a natural number and the predicate part describes an invariant that must be satisfied by $\text{ctr}$, the state variable of the system.

An initialisation may be specified as follows [20]:

\[
\begin{align*}
\text{InitCounter} \\
\text{Counter} \\
\text{ctr} = 0
\end{align*}
\]

An operation is specified in Z with a predicate relating the state before and after the invocation of that operation [20]. For example, an operation to increment the counter may be specified as follows:

\[
\begin{align*}
\text{Increment} \\
\Delta \text{Counter} \\
\text{ctr} < \text{max} \\
\text{ctr}' = \text{ctr} + 1
\end{align*}
\]
The declaration ∆ Counter means that the state Counter is changed by the operation. In the predicate, the new value of a variable is primed (\(ctr'\)), while the old value is unprimed. So the above predicate states that the new value of the counter, \(ctr'\), is the old value plus one. Note that there is an implicit conjunction (logical-and) between successive lines of the predicate part of a schema.

As well as changing the state variables, an operation may also have input and output parameters. Input parameter names are usually suffixed with ‘?’, while output parameter names are suffixed with ‘!’.

For example, the following operation for decrementing the counter has as an input parameter, the amount by which the counter should be decremented [20]:

\[
\begin{align*}
\text{Decrement} \\
\Delta \text{Counter} \\
d? : N \\
ct \geq d? \\
ct' = ct - d?
\end{align*}
\]

The next operation has an output parameter, which is the value of the counter. The declaration Ξ Counter shows that the operation cannot change the state of the Counter [20].

\[
\begin{align*}
\text{Display} \\
\Xi \text{Counter} \\
c! : N \\
c! = ct
\end{align*}
\]

Types are used to differentiate the various forms of data present in a specification [20]. Advantages of using types are that they

- Help to structure specifications by differentiating objects
- Help to prevent errors by not allowing us to write meaningless things
- They can be checked via automated tools.

Consider the example of a specification of a system used to check staff members in and out of a building. This discussion is also based on [20]. Since we shall be dealing with elements of type staff, we introduce the type Staff as a basic type:

\([\text{Staff}]\)
The state of the system is described by the following schema;

\[
\begin{align*}
    \text{Log} & \\
    \text{users, in, out} : \mathbb{P} \text{Staff} & \\
    \text{in} \cap \text{out} = \emptyset & \\
    \text{in} \cup \text{out} = \text{users}
\end{align*}
\]

The state consists of three components modelling

- the set of users of the system
- the set of staff members who are currently in and
- the set of staff members who are currently out

The predicate part of the state schema describes an invariant of the system. The invariant says that

- No staff member is simultaneously in and out.
- The set of users of the system is exactly the union of those who are in and those who are out.

An operation to check a staff member into the building is specified as follows:

\[
\begin{align*}
    \text{CheckIn} & \\
    \Delta \text{Log} & \\
    \text{name?} : \text{Staff} & \\
    \text{name?} \in \text{out} & \\
    \text{in'} = \text{in} \cup \{\text{name?}\} & \\
    \text{out'} = \text{out} \setminus \{\text{name?}\} & \\
    \text{users'} = \text{users}
\end{align*}
\]

This has an input parameter representing the member of staff to be checked in. The predicate part says that:

- The staff member to be checked in must currently be out. This is a pre-condition on the operation.
- The staff member is added to the set in.
- The staff member is removed from the set out.
• The overall set of users remains unchanged.

Similarly, an operation to check a staff member out of the building may be specified as follows:

\[
\begin{align*}
\text{CheckOut} & \\
\Delta \text{Log} & \\
\text{name?} : \text{Staff} & \\
\text{name?} \in \text{in} & \\
\text{out}' = \text{out} \cup \{\text{name?}\} & \\
\text{in}' = \text{in}\setminus\{\text{name?}\} & \\
\text{users}' = \text{users} & \\
\end{align*}
\]

An operation for removing a user’s registration, when the user is not checked in may be specified as shown below:

\[
\begin{align*}
\text{RemoveUser} & \\
\Delta \text{Log} & \\
\text{name?} : \text{Staff} & \\
\text{name?} \in \text{users} & \\
\text{name?} \in \text{out} & \\
\text{users}' = \text{users}\setminus\{\text{name?}\} & \\
\text{out}' = \text{out}\setminus\{\text{name?}\} & \\
\text{in}' = \text{in} & \\
\end{align*}
\]

Z is supported by comprehensive tool support. First of them is CADiZ [77], which is a UNIX based tool for checking and typesetting Z specifications. Zola the WYSIWYG editor is another interesting tool, which supports the production and typesetting of Z specifications. Also included are a type-checker and a Tactical Proof System (available from http://www.ist.co.uk/PRODUCTS/zola.html). Z type checkers like ZTC [74] and Z animation tools like ZANS [75] can be used to analyse Z specifications. It is projected to be compliant with the second edition of Spivey’s Z reference manual [132]. Formaliser [53] is a syntax-directed Z editor as well as an interactive type-checker, running under Microsoft Windows obtainable from Logica. Z/EVES [106] supports the analysis of Z specifications in several ways: for syntax and type checking, schema expansion, and precondition calculation.

Many methods are imposed on designers in industry by managers attempting to improve efficiency. From the feedback which has been obtained, it seems that the use of Z is one of the few specification techniques which has not been received with reluctance by industrial users [13]. The notation is gradually gaining acceptance in industry, and is taught in many computer science curricula [13].


