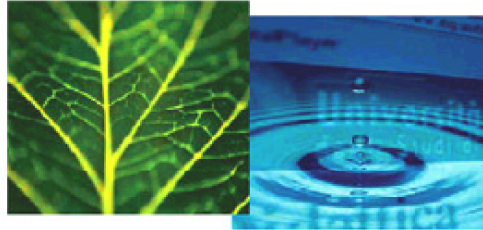


PhD Dissertation



International Doctorate School in Information and
Communication Technologies

DISI - University of Trento

SUPPORTING THE DESIGN OF SOCIO-TECHNICAL SYSTEMS
BY EXPLORING AND EVALUATING DESIGN ALTERNATIVES

Volha Bryl

Advisor:

Prof. Paolo Giorgini

Università degli Studi di Trento

March 2009

Abstract

A challenging aspect of modern information systems is the growing involvement of humans and organizations in system structure and operation. Organizational environment in which software operates, the software system itself, the related hardware components and human users are often interdependent in a non trivial way, so that it is problematic to define a system boundary. We argue that an interdisciplinary notion of a *socio-technical system* (STS) is the one that captures the above mentioned aspects. Unlike traditional computer-based system, socio-technical systems include in their architecture and operation organizational and human actors along with software and hardware ones, and are regulated and constrained by internal organizational rules, external laws and regulations. Among the challenging problems related to the analysis and design of a STS is the problem of understanding the requirements of its software component and the way in which the structure of human and organizational activities is influenced by introducing technology. In particular, an important element in the design of a socio-technical system is the identification of a set of dependencies among actors which, if respected by all parties, will fulfill all stakeholder goals, the requirements of the STS.

In this thesis, we present a framework which aims at supporting the design of socio-technical systems, specifically the design of a network of inter-actor dependencies intended to fulfill a set of initial goals. The support comes in the form of a structured process and a tool that is founded on an off-the-shelf AI (Artificial Intelligence) planner which generates and evaluates alternative assignments of actor dependencies to identify an optimal or good enough design. We explore a range of measures for evaluating optimality, inspired by AI planning, multi-agent systems, social networks and Economics. We report on the application of the framework to the domains of secure systems design, safety critical systems, dynamic reconfiguration of an STS, as well as to the problem of instantiation of STS designs. We are also experimenting with our prototype tool to evaluate its scalability to realistic design problems.

Keywords

socio-technical systems, exploring design alternatives, AI planning, evaluation metrics

Contents

1	Introduction	1
1.1	Problem formulation	1
1.2	Challenges to address	3
1.3	The approach	4
1.4	Contribution of the thesis	4
1.5	Structure of the thesis	5
1.6	Publications	5
2	State of the Art	7
2.1	Evolution of systems and design tools	7
2.1.1	Historical perspective of SE	7
2.1.2	Growing complexity of information systems	9
2.2	Socio-technical systems: introducing the concept	10
2.2.1	Socio-technical systems: definition and properties	10
2.2.2	Designing socio-technical systems: challenges	11
2.3	Modelling and analysis of socio-technical systems	13
2.3.1	Agent-oriented and goal-oriented approaches	13
2.3.2	Enterprise modelling	17
2.3.3	Designing agent organizations	18
2.4	Automation support for software design	20
2.4.1	Automatic program synthesis	20
2.4.2	Approaches in use	20
2.4.3	Software engineering as a search problem	21
2.4.4	AI techniques: planning the design - TO FINISH	22
2.5	Evaluating socio-technical designs - TO FINISH	23
3	Requirements Analysis in Tropos: Extending the Process	25
3.1	A motivating case study	25
3.2	The problem: selecting a good enough alternative	27
3.3	The proposed requirements engineering approach	28
4	Exploring the Space of Design Alternatives	31
4.1	Formalizing the input setting and desired properties of a solution	31
4.2	Planning approach	32

4.3	Desired properties of a solution	34
4.4	Planning formalization	36
4.5	Implementing the planning domain	39
4.5.1	Choosing the planner	39
4.5.2	Domain preprocessing and PDDL implementation	40
4.5.3	Complexity of the approach – TO FINISH	43
4.6	Alternative search techniques – TO FINISH	43
5	A posteriori Evaluation and Feedback Loop	45
5.1	Global evaluation criteria	45
5.1.1	Length of proposed plan	46
5.1.2	Overall plan cost	46
5.1.3	Degree of satisfaction of non-functional requirements	48
5.2	Criticality of an actor in a plan	49
5.2.1	Leaf goals satisfaction dimension	49
5.2.2	Dependency dimension	49
5.2.3	Actor criticality with respect to a set of goals	50
5.3	Local evaluation: game-theoretic insights	51
5.4	Supporting local evaluation	53
6	Validation and Scalability of the Approach	55
6.1	E-business case study	55
6.2	Other case studies - TO FINISH	60
6.3	Scalability experiments	60
7	Customizations and Applications of the Approach	65
7.1	Designing secure systems	65
7.1.1	Secure Tropos and a motivating case study	66
7.1.2	Planning domain	69
7.1.3	Case study: experimentations	72
7.2	Using risk analysis to evaluate design alternatives	72
7.2.1	A motivating case study	75
7.2.2	Planning domain	77
7.2.3	Evaluation process	78
7.2.4	Case study: experimentations	80
7.3	Runtime application: self-configuring systems	85
7.3.1	A motivating case study	87
7.3.2	Planning domain	88
7.3.3	Reconfiguration mechanism	90
7.3.4	Case study: experimentations	93
7.3.5	General architecture for self-configuring systems	95
7.4	From organizational to instance level design	98
7.4.1	A motivating case study	99
7.4.2	General schema of the approach	100
7.4.3	Planning at the organizational level: example	101

7.4.4	Organizational model instantiation	103
7.4.5	Planning at the instance level	106
7.5	Other application domains	108
8	Conclusion – TO FINISH	109
	Bibliography	111

List of Tables

4.1	Formalizing an organizational setting: predicates	32
4.2	Desired plan properties	37
4.3	Actions: preconditions and effects	38
4.4	Compared planners	40
4.5	Additional actions: preconditions and effects	42
6.1	E-business case study: plans and their evaluation	57
6.2	Experimental results: increasing the number of elementary goal trees . . .	61
6.3	Experimental results: increasing the number of goal tree levels	62
7.1	Secure systems design: primitive predicates	69
7.2	Secure systems design: actions	70
7.3	Safety critical systems design: primitive predicates	77
7.4	ATM case study: goal criticality and sat-risk	81
7.5	ATM case study: actor and goal properties	82
7.6	Self-configuring MAS: agents of the control layer	97
7.7	Formalizing the instantiated setting: predicates	105
7.8	Organizational model instantiation: steps and examples	107
7.9	Role assignment and satisfaction actions at the instance level	108

List of Figures

1.1	Sample problem: two alternative models	3
3.1	E-business case study: initial settings	26
3.2	Requirements analysis process: a general schema.	28
4.1	An extract of formalization of the diagram in Figure 3.1	33
4.2	A fragment of a plan corresponding to Figure 3.1	35
4.3	Samples of PDDL code.	41
6.1	E-business case study: adopted solution	58
7.1	Medical IS example: Secure Tropos model	68
7.2	Secure systems design: alternatives	69
7.3	Medical IS example: the planning problem in PDDL 2.2	73
7.4	Medical IS example: the chosen design alternative	74
7.5	ATM case study: air space division between ACC-1 and ACC-2	76
7.6	ATM case study: plan for increasing air space capacity	83
7.7	ATM case study: goal model for a candidate plan of Figure 7.6(b)	84
7.8	ATM case study: final problem definition and plan	84
7.9	Search-and-order MAS	87
7.10	Search-and-order MAS: planning problem formalization	89
7.11	Search-and-order MAS: initial configuration	89
7.12	Search-and-order MAS: initial configuration	89
7.13	Reconfiguration algorithm	91
7.14	Replanning procedures	92
7.15	Search-and-order MAS: first plan at t_1	93
7.16	Search-and-order MAS: second plan at t_1	94
7.17	Search-and-order MAS: first reconfiguration	94
7.18	Search-and-order MAS: first plan at t_k	95
7.19	Search-and-order MAS: second plan at t_k	95
7.20	Search-and-order MAS: third plan at t_k	95
7.21	Search-and-order MAS: second reconfiguration	96
7.22	Self-configuring MAS: 2-layered architecture	97
7.23	General schema of the approach	100
7.24	Banking scenario: formalization	102
7.25	Banking scenario: an organizational level plan	103

7.26	Banking scenario: organizational level model	104
7.27	Banking scenario: instantiation input	106
7.28	Banking scenario: (a fragment of) an instance level plan	108

Chapter 1

Introduction

In a modern world, information systems are expected to be as complex, scalable, reliable, secure and adaptable as to be able to support literally any aspect of human life, from organizing our leisure activities, to managing the operation of a whole enterprise. For many of these systems, a challenging aspect is the growing involvement of humans and organizations in system structure and operation. Indeed, organizational environment in which software operates, the software system itself, the related hardware components and human users are often interdependent in a non trivial way, so that it is problematic to define a system boundary [35]. All this calls for the new definition of a modern information system, in which its social and organizational characteristics are taken into account, as well as for methods for analysis and design of such systems. In our opinion, an interdisciplinary notion of a socio-technical system (STS) [48, 113] is the one that captures the above mentioned aspects.

Unlike their traditional computer-based cousins, socio-technical systems include in their architecture and operation organizational and human actors along with software ones, and are normally regulated and constrained by internal organizational rules, business processes, external laws and regulations [113]. Among the challenging problems related to the analysis and design of a socio-technical system is the problem of understanding the requirements of its software component, the ways technology can support human and organizational activities, and the way in which the structure of these activities is influenced by introducing technology. In particular, in a socio-technical system, human, organizational and software actors rely heavily on each other in order to fulfill their respective objectives. Not surprisingly, an important element in the design of a socio-technical system is the identification of a set of dependencies among actors which, if respected by all parties, will fulfill all stakeholder goals, the requirements of the socio-technical system.

1.1 Problem formulation

Let us make the problem more concrete. KAOS [39] is a state-of-the-art requirements elicitation technique that starts with stakeholder goals and through a systematic, tool-supported process derives functional requirements for the system-to-be and a set of as-

signments of leaf-level goals (constraints, in KAOS terminology) to external actors so that if the system-to-be can deliver the functionality it has been assigned and external actors deliver on their respective obligations, stakeholder goals are fulfilled. However, there are (combinatorially) many alternative assignments to external actors and the system-to-be. How does the designer choose among these? How can we select an optimal, or “good enough” assignment? What is an optimal assignment? The KAOS framework remains silent on such questions.

Alternatively, consider Tropos [24], an agent-oriented software engineering methodology, which uses the i^* modelling framework [126] to guide and support the system development process starting from requirements analysis down to implementation. In Tropos and i^* , goals are explicitly associated with external stakeholders and can be delegated to other actors or the system-to-be. Or, they can be decomposed into subgoals that are delegated to other actors. Thus, requirements in Tropos and i^* are conceived as *goals associated to social actors within a network of social dependencies*. In this setting, selecting a set of assignments is more complex than in KAOS because delegations can be transitive and iterative. “Transitive” means that actor A_1 may delegate goal G to actor A_2 who in turn delegates it to actor A_3 . “Iterative” means that an actor A_1 who has been delegated goal G , may choose to decompose it (in terms of an AND/OR decomposition) and delegate its subgoals to other actors.

To illustrate the problem, consider the design task in Figure 1.1 where actor A_1 has to achieve goal G , which can be refined into two subgoals G_1 and G_2 . The actor can decide to achieve the goal by itself or delegate it to actor A_2 . In both cases, there are a number of alternative ways that can be adopted. For instance, A_1 can decide to delegate to A_2 all of G (Figure 1.1b), or a part of it (Figure 1.1c). The diagrams follow Tropos modelling notation with circles representing actors, big dashed circles representing actors’ perspective, and ovals representing goals (interconnected by AND/OR-decomposition links). Social dependencies among actors for goals are represented by “De”-labelled directed links. Even for such a simple example, the total number of alternative requirements models is large, moreover, the number of alternative delegation networks grows exponentially with the number of actors as well as with the number of goals. Thus, a systematic approach and tool support for constructing and evaluating such networks of delegations would be beneficial.

However, in most software engineering methodologies the designer has tools to report and verify the final choices (be it goal models in KAOS, UML classes, or Java code), but not actually the possibility of automatically exploring design alternatives, i.e. the *potential choices* that the designer may adopt. At the same time, it is acknowledged that the latter is crucial at the early stages of the development process, namely, that “exploring alternative options is at the heart of the requirements and design processes” [87]. Therefore, automation support for the selection of alternatives could be indeed beneficial, especially taking into account that during the early development stages the design space is large, and a good choice can have significant impact on the whole project.

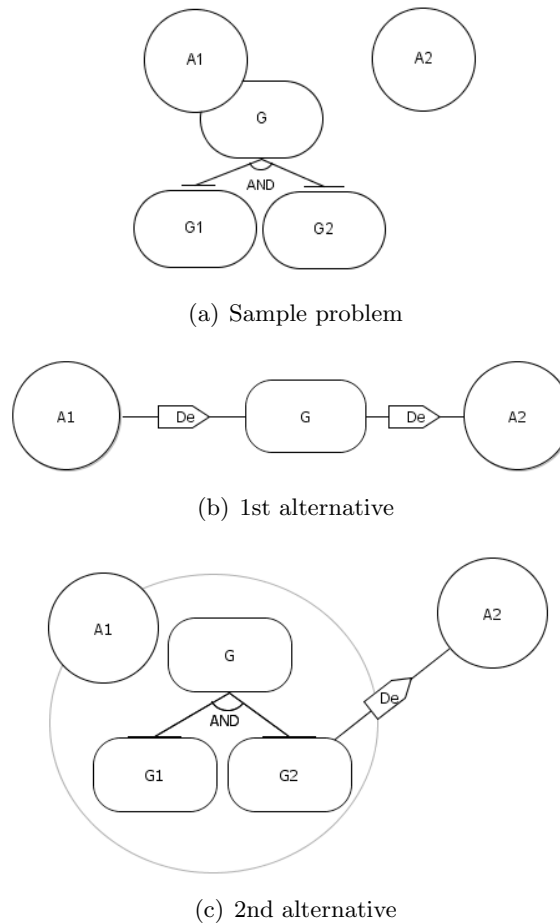


Figure 1.1: Sample problem: two alternative models

1.2 Challenges to address

The problem of exploring the space of alternative delegation networks is central to this thesis. With respect to this problem, we see the following *challenges*.

1. How can the space of alternative designs be formally represented? What techniques should be selected to perform the search in this space? What properties should a resulting solution have and how can they be guaranteed?
2. What are the optimality criteria in the context of socio-technical system design? What can be called a “good enough” solution?
3. How should the above be incorporated into requirements analysis and design processes? How can the resulting approach be evaluated?

1.3 The approach

To address the above listed challenges, we take an approach based on the observations similar to the ones reported in [37]:

...software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near optimal solution* from a large number of alternatives. Often it is not clear how to achieve an optimal solution, but it is more clear how to evaluate and compare candidates... [in search spaces, which] typically arise when a number of competing constraints have to be balanced against one another to arrive at a solution which is ‘good enough’.

We are interested in supporting the design of socio-technical systems, specifically the design of a network of inter-actor dependencies intended to fulfill a set of initial goals. The support comes in the form of a tool that is founded on an off-the-shelf AI (Artificial Intelligence) planner which generates and evaluates alternative assignments of actor dependencies to identify an optimal design. We explore a range of measures for evaluating optimality, inspired by AI planning, multi-agent systems, social networks and Economics.

Our approach solves the following problem: given a set of actors, goals, capabilities, and social dependencies, the tool generates alternative actor dependency networks on the basis of the following two steps, possibly executed in a loop.

- *Generate alternative dependency networks*: generate alternatives using a planning approach to construct an assignment of goals to actors that leads to the satisfaction of the actor goals.
- *Evaluate results and provide feedback*: evaluate alternatives by assessing and comparing them with respect to a number of criteria, provided by the designer. In case a solution is not satisfactory, provide feedback to the planner on how to constrain the further search.

This two step procedure addresses challenges 1 and 2 identified in the previous section, while challenge 3 is addressed by taking the Tropos requirements analysis and design process [24] as a starting point, and complementing it with the planning-based automation support techniques and a set of evaluation criteria and procedures.

1.4 Contribution of the thesis

The contribution of the work reported in this thesis can be summarized as follows.

- We provide both the formalization and the implementation of the design-as-planning task in the domain of socio-technical systems.
- We introduce a number of criteria evaluating the optimality of actor dependency networks, and provide the methodological guidelines on how to use these criteria.
- We define a structured tool-supported process for the requirements analysis of socio-technical systems.

- We report on the customization of the approach to several important application domains, such as secure and trusted system design, self-configuring socio-technical systems and some others.
- We validate the approach with the help of a number of case studies, and evaluate the scalability of our prototype tool to realistic design problems.

1.5 Structure of the thesis

The rest of the thesis is structured as follows.

Chapter 2 overviews the state of the art of number of the related areas. The topics include the recent trends in information system engineering; the definition, properties and approaches to socio-technical system analysis and design; approaches to automated software engineering, with the emphasis on early development phases; evaluation criteria for socio-technical systems and the associated optimization techniques.

Chapter 3 introduces a motivating case study which is used in the following three chapters for illustrative purposes, elaborates on the problem definition, and presents the general schema of the requirements engineering process which supports the designer in exploring and evaluating alternative options

Chapter 4 presents the planning based design decision support framework; in particular, the problem domain and the properties of a target delegation network are formalized and the planning domain is defined so that to satisfy these properties by construction; the implementation of the approach is discussed in details.

Chapter 5 presents an extended set of evaluation criteria related to the cost and criticality of the obtained solution, as well as to the degree of satisfaction of non-functional system requirements.

Chapter 6 illustrates the whole approach with the help of an e-business case study, discusses the other case studies to which the framework has been applied, and reports on the experimental results aimed to evaluate the scalability of the prototype tool.

Chapter 7 presents customization of the framework to the domains of secure and trusted system design, safety critical systems, self-configuring socio-technical system, as well as to the problem of instantiating socio-technical designs.

Chapter 8 summarizes the thesis and the discusses the future work directions.

1.6 Publications

Part of the results presented in the thesis were published as follows.

- Volha Bryl, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Designing Security Requirements Models Through Planning. In *CAiSE'06*, pages 33–47, 2006.
- Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing Cooperative IS: Exploring and Evaluating Alternatives. In *CoopIS'06*, pages 533–550, 2006.
- Volha Bryl and Paolo Giorgini. Self-Configuring Socio-Technical Systems: Redesign at Runtime. In *SOAS'06*, 2006.
- Yudistira Asnar, Volha Bryl, and Paolo Giorgini. Using Risk Analysis to Evaluate Design Alternatives. In Lin Padgham and Franco Zambonelli, editors, *AOSE*, volume 4405 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2006.
- Volha Bryl, Paolo Giorgini, and John Mylopoulos. Supporting Requirements Analysis in Tropos: a Planning-Based Approach. In *PRIMA'07*, 2007.
- Fabiano Dalpiaz, Raian Ali, Yudis Asnar, Volha Bryl, Paolo Giorgini. Applying Tropos to Socio-Technical System Design and Runtime Configuration. In *WOA'08*, 2008.
- Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing Socio-Technical Systems: From Stakeholder Goals to Social Networks. Accepted for publication in *Requirements Engineering Journal*.

Chapter 2

State of the Art

This section aims at providing an elaborated definition of the problem addressed in the thesis, as well as at discussing the related approaches to its solution. Specifically, we overview the historical development of software engineering approaches to understand the resulting complexity of requirements to modern information systems. Then, we introduce the notion of a socio-technical system, overview the related approaches and justify the need for the automation support for the design of these systems.

2.1 Evolution of systems and design tools

Approaches to and techniques for the development of computer-based systems have been improving in response to the changing requirements for these systems. Nowadays, high standards related to performance, security, reliability, maintainability, usability, etc. call for appropriate tools for system design, implementation and testing.

Yet another reason for the demanding and changing requirements for the system development process and techniques is that, since the beginning of computer age, the vision of what a system is and for which purpose it can be applied has changed dramatically. Compare, for example, a math computational module and autonomic (self-configuring, self-optimizing, self-healing, self-protecting) agent-based system paradigm for space exploration proposed by NASA [117].

In this section, we briefly overview the history of software systems and engineering approaches, mention a number of (relatively) recent SE trends, such as distributed and autonomic computing, and, finally, define the scope of the following discussion in this thesis to be related to the notion of a *socio-technical system*.

2.1.1 Historical perspective of SE

Let us look at the evolution of software systems, applications and development processes and tools as presented by Barry Boehm in his keynote at ICSE (International Conference of Software Engineering) 2006 [21].

In the early days of software engineering, in 1950s, the orientation towards *hardware* defined the way the development process was organized, and the main software application were aircrafts, bridges, circuits and the like. In the following decade (perhaps,

unfortunately), it was realized that software is easy to modify, which led to the adoption of “code and fix” engineering practices as opposed to scrupulous hardware-oriented design approaches. Still, the infrastructure was improving: operating systems, compilers, utilities, high-order programming languages made the life of a software developer easier. Among the systems developed, large mission-oriented applications should be noted, e.g. the Apollo manned spacecraft and ground control software.

In 1968 and 1969 NATO organized two landmark Software Engineering conferences, where, among others, the problem of defining an effective software development process was posed [100]. As a result, in 1970s, the structured programming paradigm emerged, and the related concepts were developed, such as cohesion and coupling of software modules, information hiding, abstract data types, etc. About the same time, the waterfall model was proposed by Royce [106], in which requirements engineering and design phases preceded the coding, iteration among successive development phases were introduced, and, in the later versions, verification and validation of each phase’s result were emphasized. The process was widely adopted in practice, as it was indeed advantageous for the *monolithic* systems of the time, with little user-orientation and pre-fixed (as opposed to changing) requirements.

In the 1980s, in Brooks’ famous “No silver bullet” paper [55] the main software engineering challenges were claimed to be caused by the complexity of software systems, the need of their conformity to other interfaces and/or organizational practices, as well as by software changeability and invisibility of the structure of a software system. Brooks argues that there does not exist a single solution to ease the engineering process, but it is worth paying attention to requirements refinement and rapid prototyping, careful training of the designers and, importantly, to component reuse. Indeed, in everyday activities more and more time and effort started to be saved thanks to the possibilities for software reuse offered by “more powerful operating systems, database management systems, GUI builders, distributed middleware, and office automation on interactive personal workstations” [21]. The above trend continued in the following decades with product lines architectures and COTS (Commercial, Off-The-Shelf).

Also, the 1980s have seen the emergence of standards for process compliance, Capability Maturity Models (CMM), Computer-Aided Software Engineering (CASE) tools and object-oriented paradigm in design and programming. In the 1990’s the further strengthening of object-oriented methods led to the development of UML [22] and design patterns [57].

Such phenomena as the World Wide Web, the open source software initiative, the increasing importance of legacy software and many other factors have made the requirements for software engineering processes and tools much more complicated. As a result, in the 1990s, the sequential document-driven waterfall model was replaced by the risk-driven spiral development process [19], which was intended to support concurrent engineering activities (requirements, design, implementation). Software *usability* has become of utmost importance, which caused the expansion of human-computer interaction (HCI) research area.

2.1.2 Growing complexity of information systems

During the last decades the scale and complexity of software systems have increased dramatically, and, accordingly, several new paradigms in information system architecture and development have emerged. Let us look at the most prominent of them and at the respective methods and tools.

Distributed computing

Distributed computing paradigm brought new challenges for software engineering. The motivation for the development of distributed computing is the inherent distribution of information, applications and computer users themselves. Distributed systems allow for improved availability and performance gains, but raise the issues of non-determinism, contention, synchronization, partial information and partial failure. Many standardized infrastructure solutions have been developed, such as distributed operating systems or client-server architectures (e.g. COBRA), and also, the need for effective engineering methods and tools has been recognized [85]. An important area of distributed systems research and practice is *grid computing*, which emerged in early 1990s. Grid computing is concerned with “coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations” [51].

Recent trends

By the beginning of the 21st century, software systems became as complicated as Enterprise Resource Planning (ERP) packages, data access and data mining tools, Personal Digital Assistant (PDA) solutions, etc. [21]. As seen from these examples, an important recent trend is the *integration of software and system engineering* [20], with the purpose of building user-intensive and adaptive systems. In addition, the characteristic and challenging features of both the today’s systems and engineering processes are distribution, mobility, interoperability and globalization. To cope with the new demands, Boehm proposed the updated *scalable* spiral process model [20], in which lightweight agile methods (rapid, continuous delivery of software increments, simple designs, support for changing user requirements, etc. [72]) are combined with the traditional plan-driven ones to increase the process effectiveness.

Autonomic computing

Yet another response to the growing complexity of software systems, and its consequences for the system development, integration and management, was the idea of *autonomic computing* and self-managing systems proposed by IBM in 2001 [77]. The four aspects of self-management, also referred to as “self-*” properties, are self-configuration, self-optimization, self-healing and self-protection [81]. These, according to [76], are inspired by the key characteristic features of *software agents*: autonomy, social ability, reactivity and pro-activeness [125].

Autonomic computing is aimed at minimizing human involvement, that is, having systems which exhibit goal-directed pro-active behavior, are able to perceive and react to

environment, and resolve conflicting or dangerous situation via decision making (based on the local knowledge) as well as coordination and cooperation with other systems. An architectural solution, proposed again by IBM [81], is based on the concept of autonomic element, which consists of one or several managed elements and an autonomic manager, which monitors the managed element(s), analyzes the obtained data, and then constructs and executes the corresponding plans. There exist a number of implementation of the above reference architecture, including agent-based solutions [18].

Social and organizational aspects

In addition to the distributed nature of contemporary software systems, their scale and the requirement for self-management and autonomy, their indeed challenging aspect is the growing involvement of humans and organizations in system structure and operation. Requirements engineering roadmap of the year 2000 [97] emphasized the importance of understanding the system *environment* and underlined that requirements modelling and analysis should take into account the *organizational* and *social context* in which a new system will have to operate.

This challenge is also related to the *complex system of systems trend* in modern software engineering mentioned by Boehm at ICSE 2006 [21]. The increased reliance of large scale integrated systems on the environment was identified as a RE research hotspot by Cheng and Atlee at the same conference the year after [35]. Environment in which software operates, the software system itself, the related hardware components and human users are (not trivially) interdependent, so that it is often problematic to define a software system boundary [35].

All this calls for the new definition of a modern information system, in which its social and organizational characteristics are taken into account, as well as for methods for analysis and design of such systems. We argue that an interdisciplinary notion of a *socio-technical system* (STS) is the one that captures the above mentioned aspects. We discuss its definition, properties and analysis and design issues in the subsequent sections of this chapter, and propose our own framework for the STS design decision support in the next chapters of the thesis.

2.2 Socio-technical systems: introducing the concept

In this section, we present the key characteristics of socio-technical systems, and then discuss the problems one is faced during their analysis and design as well as the related approaches.

2.2.1 Socio-technical systems: definition and properties

Unlike their traditional computer-based cousins, socio-technical systems include in their architecture and operation organizational and human actors along with software and hardware ones. The operation of a socio-technical system is regulated and constrained by internal organizational rules, business processes, external laws and regulations [113].

Some of the organizational policies and procedures, which a STS should be compliant with, may come from the legacy systems in use in an organization.

To give an example, a conference reviewing system can be viewed as a socio-technical system, as it consists of both human agents and software components, and has to conform to the rules of the reviewing process. Another example is virtual communities, which are “complex social systems enabled by a complex set of information technologies” [42]. Here the social part of the system contains various goals, organizational structures and norms, workflows and processes, whereas the technical part comprises a set of such supporting tools as mailing facilities, databases, different web applications, etc.

The notion of a socio-technical system was introduced almost 50 years ago by social scientists [48]. However, it can be viewed from two different complementary perspectives: a social sciences [48, 105, 120] and a design or engineering sciences [112, 113]. Social sciences study psychological, managerial, and organizational aspects of a socio-technical phenomenon. For example, they are interested in relationships inside workgroups, roles, supervision, motivation and the like.

Differently, the engineering perspective focuses on the design of socio-technical systems given sets of requirements, as well as their system properties. The latter perspective is the one we adopt in our work, like most of the research in software and requirements engineering [113, 92, 67].

The basic ingredients of engineering perspective can be found in Simon’s seminal vision of a Science of Design [112], where alternative designs are derived rigorously from their requirements (goals) and are evaluated in accordance with criteria that measure their effectiveness. The methodological differences in research and practice between the two perspectives are contrasted elegantly in [103].

Socio-technical systems “are imbedded in an organizational environment” [113]. Therefore, when building a STS, it is essential to understand its organizational environment and the changes technology brings to the structure of an organization, its work processes and procedures, to the responsibilities and the required skills of its human actors. In the following, we discuss the challenges of and approaches to the analysis and design of socio-technical systems.

2.2.2 Designing socio-technical systems: challenges

Developing a socio-technical system is a *systems* engineering task: not only the software should be taken into account but also hardware, system interactions with its human users and various constraints coming from organizational and social policies and regulations [113]. System engineering is inherently *interdisciplinary* involving different engineering disciplines as well as, particularly in case of socio-technical systems, organizational sciences.

Among the challenging problems related to the analysis and design of a socio-technical system is the problem of understanding the requirements of its software component, the ways technology can support human and organizational activities, and the way in which the structure of these activities is influenced by introducing technology. In a socio-technical system, human, organizational and software actors rely heavily on each other in order to fulfill their respective objectives.

Therefore, at the early stages of a socio-technical system development, there is a need for modelling and analysis techniques that help understand the organizational environment in terms of goals, interdependencies and mutual constraints of various types of actors. Such techniques can be found in the literature on agent-oriented and goal-oriented approaches as well as on organizational modelling tools for agent societies and enterprises [71, 119, 73], which we overview in Section 2.3. These approaches allow not only for modelling the organizational environment in which a socio-technical system will operate, but also provide some methodological guidelines on what models, how and in which sequence should be created, as well as the tools to check for various model properties (see e.g. [66]).

Also, in the field of requirements engineering, there exist a number of works specifically dedicated to the problem of requirements analysis for socio-technical systems [70, 67, 23, 94, 79]. For instance, [67] present a methods and a set of supporting tools for the analysis of people-oriented issues, in particular, their is on workload analysis of human actors of a socio-technical system. Based on the results of this analysis, suggestions are generated on how to partition the system requirements between automated functions, decision support, and manual procedures. The method uses an extended i^* modelling notation, and takes as input values of task completion times and complexity metrics used to calculate workloads as well as domain knowledge regarding task allocation and automation. Another i^* -based modelling and analysis method, called CREWS-SAVRE [115], aims at the analysis of dependencies between computer and human actors of a socio-technical system. They introduce coupling metrics to assess the degree of dependencies between the technological components of an STS and its users. In [70] a reference model for requirements and specification is used to analyze the (mis)behavior of an existing socio-technical system; the techniques is illustrated on the example of a chemical reactor.

The above approaches provide tools for for representing the socio-technical designs and reasoning about some of its properties. But how are these designs selected among the other alternatives? How do we identify such sets of dependencies among socio-technical system actors which, if respected by all parties, fulfill the system requirements? This constitutes a fundamental problem of socio-technical system design, the problem of multi-criteria search in the spaces of alternative design options [37]. At this point, the capacity of the human mind is not enough to cope with the complexity of the problem [111], and therefore, there is a high need for the tools able to automate (or at least to provide some support for) the process of socio-technical system design.

An important question to address while working on design automation techniques, is the one of the quality of a final design choice. Do we need the best, the optimal solution, or will a “good enough” choice suffice? As noted by Simon [112], the complexity of optimization for real-world problems is so high that even computers do not make it possible. Moreover, it is argued that “the gap between satisfactory and best is of no great importance” [112], and so identifying a *satisficing* design rather than an optimal one will do. But then, what does it mean to have a good enough design? In [37] it is argued that the problem of evaluating and comparing the available design alternatives is much easier than the optimization problem. Thus, a set of evaluation criteria, specific to the system domain and its requirements, should be be a basis for the selection of a satisficing

design among the available candidates. Some of these criteria might be based on the human expert judgements or, in general, might be difficult to formalize. Therefore, the designer should remain in the loop in order to approve or refine the options proposed by the automation procedure.

In Section 2.4 we overview the existing design automation techniques, including AI planning algorithms, which we adopt in this thesis to support exploring the space of design alternatives. Then, in Section 2.5 we discuss the approaches related to the evaluation of socio-technical system designs with respect to their structural properties, non-functional characteristics and some others.

2.3 Modelling and analysis of socio-technical systems

In this section, we discuss the agent and goal-based modelling and analysis techniques, as well as modelling methodologies for agent societies and enterprizes.

2.3.1 Agent-oriented and goal-oriented approaches

In modelling and analyzing requirements for information systems and organizations, two important and interconnected trends are agent [125, 71] and goal-orientation [119]. In the following, we overview a number of related approaches, which allow for modelling and reasoning about a socio-technical design in terms of agents and their strategic interests, agent coordination and negotiation, commitments and obligations, institutions and norms.

***i**/Tropos**

The *i** modelling framework [126] offers primitive concepts of (social) actors, goals and actor dependencies, which allow for modelling both software systems and organizational settings. The framework includes the *strategic dependency model* for describing the network of inter-dependencies among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors.

According to [126], the *strategic dependency model* provides an intentional description of a process in terms of a network of dependency relationships among actors. It aims at capturing the underlying motivations and intents behind the modelled process. The model helps the analyst to identify stakeholders, analyze opportunities and vulnerabilities, and recognize patterns of relationships, such as various mechanisms for mitigating vulnerability. The *strategic rationale model* provides an intentional description of process in terms of process elements and rationale behind them. Unlike strategic dependency model, which focuses on the external dependencies among actors, the strategic rationale model describes the intentional relationships that are “internal” to actors, such as means-ends relationships that relate process elements, providing explicit representations of “why” and “how” and alternatives.

Tropos [24] is an agent-oriented methodology which uses extended *i** notation with *actor*, *goal*, *sofgoal*, *task*, *resource*, and *dependency* as basic modelling constructs. An *actor* is an intentional entity that performs actions to achieve goals. A *goal* represents

an objective of an actor, while a *softgoal* is a goal for which there is no explicit criteria on whether it is satisfied or not (e.g. “improve system usability”). A *task* specifies a particular sequence of actions that should be executed for satisfying a goal, and a *resource* represents a physical or an informational entity.

One of its key ideas is to use the same concepts throughout the whole system development process, from early requirements analysis to implementation. This aims at reducing the existing semantic gap between the technical (e.g., classes, objects, tables, functions, etc.) and social, or organizations (e.g., roles, strategic goals, stakeholders, etc.) aspects of system development. As explained in Chapter 1, the planning-based framework proposed in this thesis, has originated in context of Tropos, and uses its notation and modelling principles. Therefore, in the following we provide details on phases of Tropos methodology.

During the *early requirements analysis* stakeholders along with their goals are identified. In this phase, actor diagrams and rationale diagrams are used. An *actor diagram* is a graph of actors interconnected with strategic dependencies for goals. A *dependency* represents an “agreement” between two actors, the depender and the dependee, on the delivery of a dependum. The dependum can be a goal/softgoal to fulfill, a task to perform, or a resource to deliver. Graphically, actors are represented as circles, and goals, softgoals, tasks and resources are, respectively, represented as ovals, clouds, hexagons, and rectangles; dependencies among actors have the form depender \rightarrow dependum \rightarrow dependee. As the analysis proceeds, actor diagrams are refined and extended. In a *rationale diagram* the goals of a specific actor are analyzed and dependencies with other actors are established. Goals are decomposed into AND/OR-subgoals and positive/negative *contributions* of subgoals to other goals are specified. During *late requirements analysis*, a new actor, the system-to-be, is included into the organizational model, and its dependencies with the existing actors are analyzed. During the *design* phase, which is subdivided into architectural and detailed design, the capabilities the system actors need to fulfill their goals are identified, a set of agent types is defined and each of them is assigned one or more different capabilities, finally, agent micro level is specified.

Tropos framework also provides the designer with a set of reasoning tools for requirements analysis, validation and verification (e.g. goal reasoning tools [65], automatic verification of security and trust requirements in Secure Tropos [66]). Based on organizational theory and i* framework, a number of *organizational styles* and *social patterns* were proposed [56, 83] to guide the development of a socio-technical system. Organizational styles describe the overall structure of the organizational context of the system or its architecture, while social patterns focus on the social structures necessary to achieve one particular goal.

KAOS

KAOS [39] is a goal-oriented approach, which aims at formal modelling of functional and non-functional system requirements. KAOS methodology provides a specification language for (i) capturing *why*, *who*, and *when* aspects in addition to the usual *what* requirements; (ii) goal-driven elaboration; and (iii) providing meta-level knowledge used for local guidance during method enactment.

The language provides a rich ontology for capturing requirements in terms of *goals*, *constraints*, *objects*, *actions*, *agents*, *events*, etc. Links between requirements are represented as well to capture *refinements*, *conflicts*, *responsibility assignments*, etc. Modelling goals of different types is supported, which allows defining goal attributes, links between goals (e.g. to model the situation when a goal negatively or positively supports other goals), AND/OR goal refinement links, and links between goals and agents.

UML-based approaches

AUML (Agent UML) [14], being a modelling language rather than a methodology, allows for the modelling of agents, their internal behavior and interactions within an organization. The key idea of Agent UML is to reuse those UML diagrams that fit the need of MAS designers and to extend UML (e.g. through stereotypes, tagged values, constraints) in order to represent agents. Two new modelling facilities introduced in AUML are *sequence diagrams* and *agent class diagrams*. Sequence diagrams define the exchange of messages through protocols, and comprise agents and agent roles, connectors, messages and message conditions, protocol templates, etc. Agent class diagrams illustrate the static design view of a system with a set of classes, interfaces, collaborations and their relationships.

MESSAGE modelling language [31] is based on RUP (Rational Unified Process) notation that is, in turn, based on UML. MESSAGE extends UML with agent-related concepts such as *agent*, *organization*, *role*, *goal*, *task*, *interaction* and *interaction protocol*. An organization is defined as a group of autonomous agents working together for a common purpose. The distinction between role and agent is analogous to that between interface and class: a role describes the external characteristics of an agent in a particular context. The MESSAGE methodology covers the analysis and design phases of a multi-agent system development, in which it follows the iterative and incremental approach of RUP.

GAIA

GAIA [127] is the first agent-oriented software engineering methodology that explicitly takes into account social concepts. One of the key concepts in GAIA is that of an *organization*, which is viewed as a collection of *roles*, which are, in turn, defined in terms of *responsibilities*, *permissions*, *activities* and *protocols*. Responsibilities define the functionality of the role, while permissions are the “rights” which allow the role to perform its responsibilities. Activities are computations that can be executed by the role along, and protocols define the interaction between roles. An organization can be subdivided into *suborganizations*, and one agent can be involved in multiple organizations. In an organization, an agent can play one or more *roles*, each associated with a set of expected behaviors. To accomplish their roles, agents typically need to interact with each other to exchange knowledge and coordinate their activities. The notion of an *environment* of a multi-agent system is related to all the entities and resources that the system can exploit, control, or consume.

With respect to designing a new organization, two important notions are *organizational rules* (the constraints that the actual organization, once defined, will have to respect) and an *organizational structure*. The GAIA design process starts with the analysis phase,

which aims at defining an environmental model, preliminary roles and interaction models, as well as the set of organizational rules. Then, during the architectural phase, the system organizational structure is defined, which is followed by a implementation-independent multi-agent system specification at the detailed design phase.

AORML

RAP/AOR (Radical Agent-Oriented Process/Agent-Object-Relationship) [69] agent-oriented software engineering process follows the RUP, but is based on AORML (AOR modelling language) instead of object-oriented modelling approach. RAP/AOR does not aim at capturing human-like intelligence features such as desires and intentions, or sophisticated forms of proactive behavior. Rather, its focus is on declarative models of communication and interaction founded on reactive behavior and on the basic mental state components of beliefs, perceptions, and commitments. In RAP/AOR, a problem domain is modelled from the *interaction*, *information*, and *behavior* viewpoints.

AORML distinguish active and passive entities, that is, agents and (non-agentive) objects of the real world; an entity can be an agent, an event, an action, a claim, a commitment, or an ordinary object. Agents can communicate, perceive, act, make commitments, and satisfy claims. Objects are passive entities with no such capabilities. In addition to human and artificial agents, AORML also includes the concept of institutional agents (e.g. organizations or organizational units), which are composed of a number of other agents that act on their behalf. An important behavior modelling element of AORML is *reaction rules*, used to express interaction patterns. There are two basic types of AOR models, which differ with respect to the position of the observer of a system: *external* models for representing a (business) domain and *internal* models for specifying a multi-agent system design.

MAS-CommonKADS

MAS-CommonKADS [78] is an agent-oriented software engineering methodology that guides the process of analysing and designing multi-agent systems. The origins of MAS-CommonKADS come from CommonKADS, a knowledge engineering methodology, and from object-oriented methodologies, such as Object Modelling Technique, Object-oriented Software Engineering and Responsibility Driven Design.

The functional requirements of a multi-agent system are described via a set of models the properties of *agents*, *tasks* they carry out, *expertise model* they need to achieve their goals, *organizations* they form or interact with, agent *coordination* and their *communication* with human users. The design model consists of a network design model, which concern the relevant aspects of the agent network infrastructure, an agent design model, which concern identifying the most suitable agent architectures, and platform design, which concern selecting the agent development platform for the architecture of each agent.

2.3.2 Enterprize modelling

The problem of organizational design has been widely studied in both information and social sciences, which makes a thorough overview of the field a challenging task. Thus, only two subareas of organizational design are considered in the thesis. The following section reviews briefly the area of enterprise modelling, i.e. the organizational modelling of enterprises. Then, in the next section, we discuss a number of approaches to the modelling and design of agent organizations.

Enterprise Project

The aim of the *Enterprise Project* [114] is to provide a set of supporting tools (Enterprise Tool Set) for modelling business activities of an organization. The components of the Enterprise Tool Set include Process Builder, used to model business processes in an organization, Agent Toolkit, an agent-based solution to support the integration of the tools already in use in an organization, Task Manager, an interface between the user and the Tool Set, and Enterprise Ontology, which underlies the integration and communication of the Tool Set components.

Enterprise Engineering Methodology

The *Enterprise Engineering Methodology (EEM)* [10] is a part of the PRIDE framework, which is a collection of methodologies and techniques for Information Resource Management. The EEM and requirements engineering for socio-technical systems have similar objectives: EEM aims at developing the Enterprise Information Strategy (EIS), which is a plan to satisfy the information needs of an enterprise (e.g. by developing the new or modifying the existing information systems). This plan is synchronized with business needs of an enterprise and is based on the modelling and analysis of the enterprise and its environment.

The EEM comprises the following five phases: *project planning*, in which an organization is defined in terms of its objectives and relations with the environment; *logical enterprise analysis*, in which the logical structure of an organization, its operation and resources is defined; *physical enterprise analysis*, which focuses on the definition of the physical structure of an organization; *development of the EIS*, which defines the objectives and the plan an organization will follow, and should be reviewed routinely to align it with the changing business requirements and conditions; *evaluation*, which is about initiating and estimating the impact of the developed EIS.

CIMOSA

The *Computer-Integrated Manufacturing Open-System Architecture (CIMOSA)* [5], developed by the AMICE Consortium, aims at efficient and adaptive enterprise operation supported by information technology. CIMOSA defines an integrated methodology to support all phases of a CIM system life cycle from requirements specification through system design, implementation, operation, and maintenance. An organization is modelled

according to the following four views: *function* view, which describes the functional structure required to satisfy the objectives of an enterprise and the related control structures; *information* view, which describes the information required by each function; *resource* view, which describes the resources and their relations to functional, control and organizational structures; *organization* view, which describes enterprise organizational structures in terms of responsibilities assigned to individuals for functional and control structures, information and resources.

GERAM

The *Generalised Enterprise Reference Architecture and Methodology (GERAM)* [16] is meant to be not yet another proposal for an enterprise reference architecture, but a generalizing framework, applicable to different types of enterprises, in which the existing enterprise integration knowledge can be organized. GERAM provides a description of all the elements recommended in enterprise engineering and integration, thereby it sets the standard (i.e. defines the criteria to be satisfied) for the collection of tools and methods which support the initial enterprise integration design as well as change and adaptation during the enterprise operational lifetime.

GERAM proposes the *methodologies for enterprise engineering* and the *modelling languages*, which are supported by *enterprise engineering tools*. Modelling languages are used by the methodologies to model the structure, content, and behavior of an enterprise. Enterprise models are considered to be an essential component of enterprise engineering and integration. Modelling languages allow for modelling human activities within an enterprise as well business processes and the supporting technology. In GERAM, enterprise models represent the wide range of enterprise operations, including its manufacturing or service tasks, organization and management structures, and control and information systems. The modelling process is supported by using *partial models* which are reusable models of human roles, processes and technologies.

2.3.3 Designing agent organizations

In this section, we overview a number of approaches related to agent organizations [73]. The concept of social and organizational *agent* as well as the related constructs and structures are essential for capturing socio-technical system properties.

OperA

OperA methodology [44] focuses on designing an open organizational environment in which autonomous intelligent adaptive agents act and interact. In order to model roles, goals and interactions within an organization, OperA proposes a *3-layered approach* which comprises the following models. *Organizational* model describes the desired or intended behavior and overall organizational structure of agent society in terms of roles, communication and coordination models and norms. *Social* model populates the organizational model with specific agents mapped to roles through a social contract. Social contracts describe the

agreed behavior of an agent within the society in terms of externally observable events. *Interaction* model describes agent interactions by the means of interaction contracts.

A formal theory for the OperA framework is the language for contract representation (LCR), based on deontic temporal logic. LCR allows describing and verifying contracts that specify interaction between agents.

AGR

The *AGR* model [49] (the old name is AALAADIN) is based on three core concepts: *agent*, *group* and *role*. An agent is specified as an active communicating entity trying to achieve its design goals, no constraints are imposed on the internal architecture of agents. A group is defined as a set of agents playing specific roles, where role is an abstract representation of an agent function within a group. Each agent can be part of one or more groups and play one or more roles. Agents can interact by sending messages only when they belong to the same group. The core agent-group-role meta-model allows for modelling different forms of agent organizations (e.g. hierarchical or market-like) in different application domains.

To analyze and design multi-agent organizations the following two structures are used: *group structure* is an abstract description of a group, which specifies the roles within a group and interactions among them; *organizational structure* is a set of group structures, which specifies the interconnections between agents, roles and groups. The MadKit agent-based platform implements the AGR concepts, and follows three design principles: micro-kernel architecture (i.e. a minimal set of facilities allowing for the deployment of agent services), agentification of services (i.e. system services are represented as agents playing specific role in meta-level groups), and graphic component model. Any services beside those assured by the micro-kernel are handled by agents, organized in groups, and identified by roles.

MOISE+

MOISE+ (Model of Organization for multi-agent SystEms) [74] represents a multi-agent organization in terms of its structure, functioning and the deontic relation among these two. The analysis of an organization along each of the dimensions can proceed independently. In *MOISE+*, the specification of an organization is formed by *structural specification*, *functional specification* and *deontic specification*.

Structural specification is based on the concepts of a *role*, *role relation* and a *group*. A role is a set of constraints that an agent should follow when it enters a group and commits to playing that role. A relation, or a link among roles implies certain permissions granted to its participants, e.g. in case of a *communication* link, the agent playing the link source role is allowed to communicate with the agent playing the link destination role. At the level of groups, *compatibility* constraints among two roles are introduced, which state that the agents playing one role are also allowed to play the other role. Roles, role relations and groups are used to build, respectively, the individual, social and collective levels of an organizational structure. *Functional specification* in *MOISE+* is based on the concepts of missions (a set of global goals) and global plans, assembled in a social scheme, which

is a goal decomposition tree where the responsibilities for the sub-goals are distributed in missions. *Deontic specification* relates structural and functional specifications of an organization with the help of *permissions* and *obligations* of a role on a mission.

2.4 Automation support for software design

...

2.4.1 Automatic program synthesis

Almost fifty years ago the idea of deriving the code directly from the specification (such as that advocated in [91]) started a large programme for *deductive program synthesis*, which is based on the following idea. A system goal together with a set of axioms are specified in a formal specification language. Then the system goal is *proved* from the axioms using a theorem prover. A program for achieving the goal is extracted from the proof of the theorem. High expectations for this approach had even led to the definition of the *transform software engineering process model* [13], which underlying assumption is the existence of the automatic transformation of the formal specification of a software into the a program satisfying the specification.

A number of software tools were developed to support the automated program synthesis, e.g. Specware automated software development system [1, 123], Amphion [89, 104], a domain-oriented design environment developed at NASA Ames Research Center, Nuprl proof development system [38, 32], and the like. The field is still active now, with the papers reporting on applications of and extensions to the existing techniques [123, 32, 104], or proposing the new ones [47, 93]. However, these approaches are largely domain-specific, and in some cases do not actually guarantee that the synthesized program will meet all requirements stated up front [47]. Existing frameworks often require from their users a considerable expertise in formal mathematics: it is acknowledged that “users without a background in formal mathematics find that developing a formal problem specification is usually more difficult than developing code manually” [89].

2.4.2 Approaches in use

...

Model driven software development

Another approach to facilitating the work of the designer is supporting the tedious aspects of software development by automating the design refinement process. This approach underlies Model Driven Architecture (MDA) approach [98], which has been proposed by Object Management Group and is a framework for defining methodologies for software system development. MDA aims at bridging the gap between problem and implementation domains, and, therefore, its central focus is on (possibly automatic) model transformation, for instance, from the platform-independent model of the system to platform-specific models used for implementation purposes. Models are usually described in UML, and

the transformation is performed in accordance with the set of rules, called mapping. Transformation could be manual, or automatic, or mixed. Tools supporting MDA exist and are used in the Rational Unified Process (RUP) for software development. Yet, the state-of-the-art MDA technologies provide support for storing and manipulating models rather than for rigorous model transformation and analysis [53].

Design patterns

A widely accepted proposal of Gamma et al. on design patterns [57] aims at supporting the process of software system design by proposing template solutions to recurrent design problems. Namely, a design pattern is a solution (commonly observed from practice) to a certain problem in a certain context, so it may be thought as a problem-context-solution triple. Several design patterns can be combined to form a solution. Notice that it is still the designer who makes the key decision on what pattern to apply to the given situation.

Numerous proposals for design patterns for different domains and different steps of the software development process can be found in the literature, including organizational patterns for Tropos early requirements analysis [83].

2.4.3 Software engineering as a search problem

There exist a number of works (e.g. see [37] for an overview) in which various problems a software engineer is faced with during the development cycle are treated as *search problems*. Indeed, designing a software system means exploring the space of alternative options in order to find an optimal solution, normally a trade-off between the development costs and the quality of a resulting artifact.

Clarke et al [37] overview the application of such well-known search techniques as hill climbing, simulated annealing, tabu search, genetic algorithms, genetic programming to test data generation, module clustering, cost estimation, etc. In automatic search-based test data generation the aim is to construct such a set of test cases that the code (or specification) coverage is maximized. In cost estimation, genetic programming is used to learn the cost predictive function from a training data set.

Another interesting application of search-based techniques, which objectives are related to those of the planning-based framework presented in this thesis, is the search-based requirements analysis and optimization [129, 50]. In essence, requirements optimization is about finding “the ideal set of requirements that balance customer requests within resource constraints” [129]. This problem is particularly relevant in the context of iterative development, where it is also known as the *next release problem*: which requirements (and of which customers) to select to implement in the next release. Requirements engineering for multiple customers, which often have competing or even conflicting needs, can be naturally framed as a search problem, and in [129, 50] it is proposed to use evolutionary algorithms to handle it.

In particular, in [50] a notion of *fairness* of requirements assignments is viewed as a fitness function. That is, in a *fair* requirements assignment each customer gain roughly equal value or spend equal cost. Therefore, the aim of a search-based algorithm is to

maximize the fairness of requirements assignment, after the values/costs are associated to each requirement for each customer.

The authors of [50] consider different notions of fairness at the same time, and thus, the fitness function is multi-objective. Here an important notion is a set of non-dominated solutions, also called Pareto-optimal front. A set solution is said to be dominated by another solution if the latter is worse or equal than the former with respect to all objectives and strictly worse with respect to at least one objective. For example, if our objective is to minimize the development time and cost, then the set of requirements which implementation requires 10 days and 100 cost units is dominated by the set which implementation requires 8 days and 90 cost units, and the latter is dominated by the set which implementation requires 8 days and 85 cost units. The set of solutions each of which is not dominated by any other solution, is called Pareto-optimal front, and the aim of the search algorithm is to identify it.

The authors also argue that multi-objective search problems are a natural fit for requirements optimization, as often several (possibly interdependent) factors should be optimized independently, e.g. cost and value, or implementation-based versus business-based objectives, or simply different types of resources needed for the implementation of a requirement. However, the most tricky point in the real life application of the (multi-objective) search based techniques is the need to *quantify* these objectives, that is, define numeric values and costs associated to each requirement from a viewpoint of each customer.

In general, the application of search-based techniques to requirements optimization seems a promising area, though a number of challenges are to be addressed [129]. For instance, it is not always obvious how to define and/or calculate a fitness function, how to chose “the best” search algorithm and initialize its input parameters. Among other issues are scalability, explanation the results to customers, handling requirements dependencies, etc.

2.4.4 AI techniques: planning the design - TO FINISH

AI (Artificial Intelligence) Planning is about automatically determining a course of actions (i.e., a plan) needed to achieve a certain goal where an action is a transition rule from one state of the world to another. Planning is useful in the situations where it is not feasible to enumerate in advance the number of possible transitions from the initial to the desired state [8]. It has found a number of applications (robotics, process planning, autonomous agents, etc.).

To define a planning problem, one need to specify

- the initial state of the world,
- the desired state of the world,
- the actions that can be performed.

Once the domain is defined, the solution to the planning problem is a (not necessarily optimal) sequence of actions that makes it possible to reach a desired state starting from the initial state.

There exist several ways to represent the elements of a classical planning problem, i.e. the initial state of the world, the goal of the planning problem (i.e. or the desired state of the world), and the possible actions system actors can perform. PDDL (Planning Domain Definition Language) is the widely used specification language proposed in [63]. In the implementation of our approach, we use PDDL of the version 2.2 [46], which supports, among others, derived predicates and timed initial literals.

There are two basic approaches to the solution of planning problems [122]. One is graph-based planning algorithms in which a compact structure, called Planning Graph, is constructed and analyzed. In the other approach the planning problem is transformed into a SAT problem and a SAT solver is used.

A few works can be found which relate planning techniques with information systems requirements analysis and design [6, 58, 33]. For example, one of the early proposals [6] describes a program called ASAP (Automated Specifier And Planner), which automates a part of the domain-specific software specification process. ASAP supports a designer in selecting methods for achieving user goals, discovering plans that result in undesirable outcomes, and finding methods for preventing such outcomes. The disadvantage of the approach is that a designer still performs a lot of work manually when determining the combination of goals and prohibited situations appropriate for the given application, defining possible start-up conditions and providing many other domain-specific expert knowledge.

Castillo et al. [33] present an AI planning application to assist an expert in designing control programs in the field of Automated Manufacturing. The system they have built integrates POCL, hierarchical and conditional planning techniques (see [33, 101] for references). The authors consider standard planning approaches to be not appropriate with no ready-to-use tools for the real world, whereas in our paper the opposite point of view is advocated. An application of the planning approach to the design of secure systems is proposed by Gans et al. [58]. Their work is based on i^* modelling approach [126] and ConGolog [64], a logic-based planning language. However, the authors focus more on representing/modelling trust in social networks, than on automating the design, and do not go far in explaining how they exploit the planning formalism.

2.5 Evaluating socio-technical designs - TO FINISH

(1) Overview of the evaluation criteria coming from organizational theory, economics, SE and RE practices (e.g. NFR classification and analysis techniques), (social) networks literature (e.g. the notion of criticality).

A body of work known as value-based software engineering [17] is focused on the definition and application of value-driven methods aimed at supporting the search for an optimal engineering solution.

Vladimir Tomic, the work on business value.

(2) Game theoretic perspective of optimality (including mechanism design).

Game theory is an established discipline which deals with conflicts and cooperation among rational independent decision-makers, or players. The key concept in classical game theory is the notion of equilibrium [99] which defines the set of strategies, one for each

player, which none of the independent rational players wants to deviate from. By playing an equilibrium each player maximizes his utility locally, given some constraints. For example, playing the Nash equilibrium means that no player can benefit when deviating from his equilibrium strategy given that all other players play the equilibrium.

Game theory is applied in various areas, especially in economics (modelling markets, auctions, etc.), corporate decision making, defense strategy, telecommunications networks and many others. Among the examples are the applications of game theory to so called network games (e.g. routing, bandwidth allocation, etc.), see [116] for references.

(3) Evaluating and comparing alternative designs (models) in current RE/SE approaches.

Recently, the problem of evaluating and comparing requirements models has received a lot of attention, and there are several proposals in the literature which address the problem from different, often complementary perspectives. For example, a number of works ([36, 65, 121]) analyze the contribution links between goals and non-functional requirements in order to compare the alternative choices with respect to such non-functional requirements as security, efficiency, user-friendliness, etc. In [54] a problem of defining quantitative evaluation metrics for evaluating i^* models is discussed, with predictability of model elements as an example of the measured property. In AGORA requirements elicitation framework [80], each stakeholder has a preference matrix for each of the goals, which contains not only his preference value for the goal, but also his estimation of the preference values for this goal for other stakeholders. These matrices can aid a designer in selecting and adopting a goal from various alternative by identifying and analyzing goal conflicts. Yet another i^* -related proposal [115] uses coupling metrics to assess the degree of dependencies between the system and the users.

Chapter 3

Requirements Analysis in Tropos: Extending the Process

In the following we present an e-business case study, and, with the help of it, discuss the problem of exploring and evaluating the space of design alternatives in the context of Tropos. Then, we present an overall schema of the requirements engineering process which addresses the above problem.

3.1 A motivating case study

We further illustrate the problem on the basis of an e-business case study, adapted from the SERENITY EU project¹. The case study focuses on the banking domain, namely, on the loan provision process. A detailed description of the case study is given in [110], while here for the sake of clarity and ease of understanding, we present only those details which are relevant to the paper.

The main actors of the e-banking scenario are the *customer*, the *BBB Bank* (hereafter *the bank*) and the *credit bureau*. The customer has an intention to buy a house, and to do this, she needs to get a loan from a bank. When looking for a suitable proposal, the customer not only checks whether or not she is granted a loan (which might depend, e.g., on her employment status), but also on the conditions upon which the loan is given (e.g. the loan costs).

The diagram in Figure 3.1 presents the actors of the banking scenario along with their high-level goals and inter-dependencies. In the diagram, an arrow from *get a loan* to *find money* represents a means-end relation between two goals, meaning that whenever the source goal (means) is satisfied, the target goal (end) becomes satisfied.

Although the bank has a complex organizational structure and contains various departments, management hierarchy, and hundreds of employees, we consider only three roles (a role is a specification of an actor) within the bank played by its employees. In particular, we model and analyze strategic interests and functionalities of the bank in terms of those of the *bank manager*, the *junior clerk* and the *senior clerk*, which are represented as roles in Figure 3.1 and are connected to *BBB bank* actor with is-part-of

¹<http://www.serenity-project.org/>

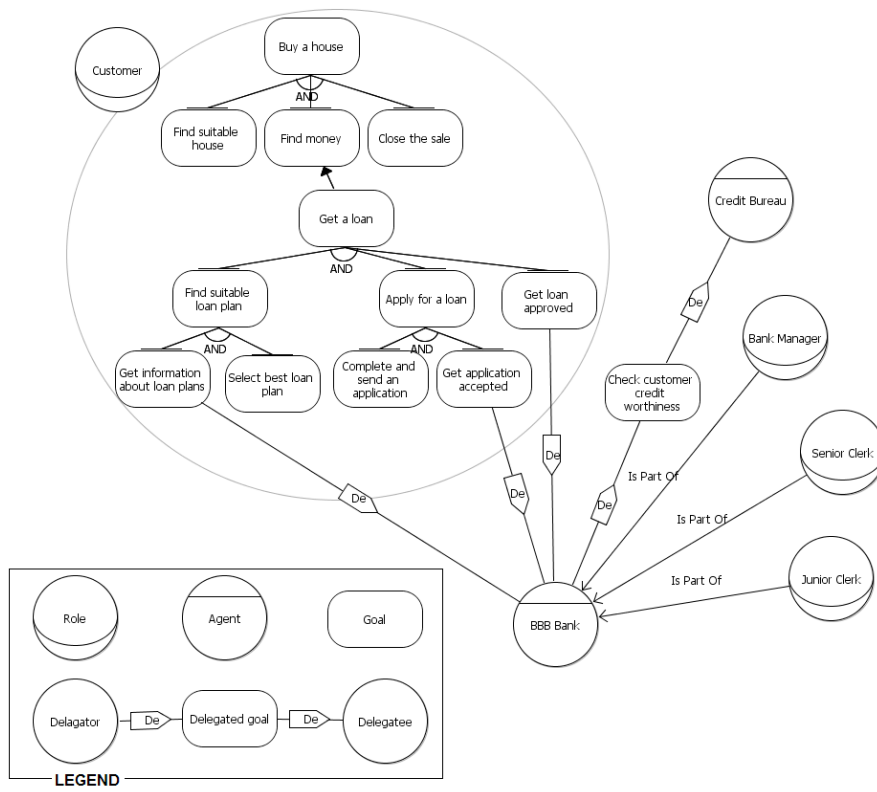


Figure 3.1: E-business case study: initial settings

relation. The manager's duty is leading the (local agency) of the bank. He can be involved in all steps of the loan approval process, but usually he is only involved when a final decision or supervision is needed. The junior clerk is a bank employee with just a couple of years of working experience, therefore he can deal only with those activities which require less skill and responsibility. The senior clerk is an experienced bank employee, who is responsible for performing all banking transactions for the customer.

The *credit bureau* is "a third-party business partner of a financial institution that processes, stores and safeguards credit information of physical and industrial agents" [110]. The bank, through the senior clerk or the bank manager, can contact the bureau in order to obtain the information about credit worthiness of the customer.

3.2 The problem: selecting a good enough alternative

According to Tropos methodology, the process of modelling and analyzing requirements for a socio-technical system is the following. Starting with the initial organizational setting presented in the diagram in Figure 3.1, the designer refines the goals and relations between the actors/goals in order to provide the details on how the loan approval is organized and how work is divided among existing and/or new actors. In particular, goal models are constructed for all top-level goals (*get a loan* in our case). A goal model [65] is a decomposition tree of a root goal into AND- or OR-subgoals. AND-decomposition of a goal refines the goals into subgoals which are to be satisfied in order to satisfy the parent goal, while OR-decomposition of a goal consists of a list of alternative subgoals any of which can satisfy the parent goal.

Given a goal model, the question is the following: *what is an optimal or, at least, good enough assignment of leaf goals to actors so that the root goal is satisfied when all actors deliver on the goals assigned to them?* The number of possible assignments is limited by the fact that different actors have different capabilities, and not all actors can delegate to all others. That is, an actor can delegate a goal only to actors he can depend on, e.g. those he knows, or manages, or pays money to. Still, the number of alternative delegation networks grows exponentially with the number of actors and/or goals that need to be dealt with. This calls for a way to automatically construct and evaluate possible alternatives to facilitate the design process.

To give an example, the corresponding problem to be addressed in the case study consists of finding the most effective way for bank employees to collaborate in order to satisfy customer requests and bank strategic interests. There could be several alternatives with respect to goal decomposition and assignment, depending on, for example, the involvement of the bank manager, or the division of labour between the junior and senior clerks. Another source of alternatives is the possible automation of (part of the) banking procedures. For this purpose, another system actor is introduced, the *bank internal computer system* (hereafter *the system*), which is capable of storing data, performing different calculations (e.g. of loan costs), providing contract templates, etc. Different decisions on what procedures to automate produce alternative social networks, which are all intended to satisfy customer needs but differ in terms of cost, risk, workload, etc.

The problem of exploring the space of alternative delegation networks is central to

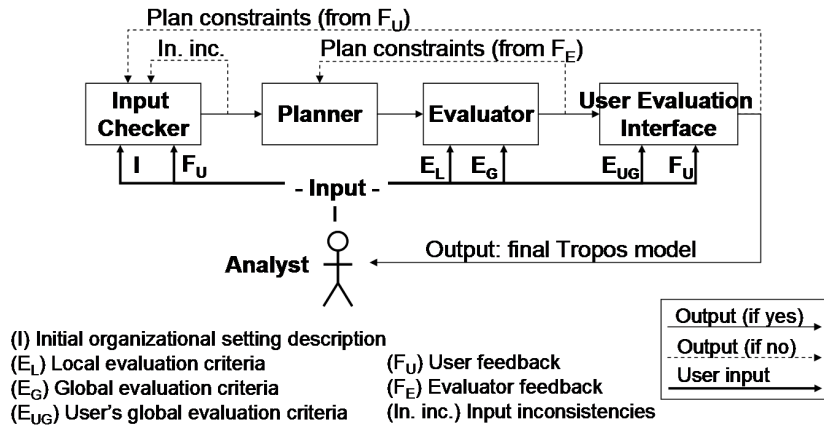


Figure 3.2: Requirements analysis process: a general schema.

this paper. To address it, we complement the Tropos requirements analysis and design process as described in the following section.

3.3 The proposed requirements engineering approach

Our proposal is to structure the requirements analysis process to support a designer in constructing and evaluating requirements models. The general schema of the process, which we have already presented in [28], is presented in Figure 3.2. The description of an initial organizational setting is provided by a designer, and includes actors, their goals and capabilities, dependencies among actors, possible ways of goal refinements (decompositions of a goal into AND/OR-subgoals), and other goal and actor properties (see Section ?? for details). This input is analyzed and iteratively improved so as to output a model that guarantees the fulfillment of stakeholder goals and is good enough with respect to a number of user-defined criteria. Most of the process steps can be automated. However, the presence of a human designer (referred to as an analyst in the schema) is inevitable: the design of socio-technical systems can be supported by tools but cannot be automated.

As a first step, the process checks whether there exists at least one assignment of goals to actors that leads to the satisfaction of top-level goals. **Input checker** analyzes the organizational setting description, detects inconsistencies, and proposes possible improvements, which then are approved, or rejected, or modified by the designer. In particular, it is checked whether available actors possess enough capabilities to collectively satisfy their goals, and whether the relationships between actors permit this to happen. To analyze actor capabilities means to check that for each goal it is possible to find an actor who is capable of achieving each of its AND-subgoals or at least one of its OR-subgoals. To analyze actor interdependencies means to check whether a goal can be delegated from an actor who wants to achieve it to an actor who is capable of achieving it, namely, whether there exists a path between two actors. In [28], we give details on analyzing and dealing with missing capabilities, which is based on label propagation algorithm similar to the

one presented in [65]. After a missing capability is detected, there are two ways to deal with it:

- Add a new capability to an existing actor. Such a decision could be based on the actual capabilities of this actor. Namely, if this actor is already capable of achieving one or several goals of the same type, it is likely that it could manage the new goal as well.
- If there is no way to add the missing capability to one of the existing actors, a new actor might be introduced.

After the input is checked, the first possible alternative is generated by the **Planner**, which exploits AI planning algorithms [122] to construct a social network that is capable of achieving the specified high-level system goals and, at the same time, satisfies a number of (optimality) criteria (see Chapters 4-5 for the details).

An alternative generated by the planner is then assessed by the **Evaluator** with respect to a number of criteria. Some optimality criteria can be incorporated into the planning domain formalization (see Section 4.3), while the others are used a posteriori to evaluate an already obtained solution in terms of costs, risks, etc. These criteria are defined by the designer and refer to the optimality of the solution either from a global perspective (e.g. assessing the system security or efficiency), or from the local perspectives of stakeholders (e.g. assessing the workload distribution). Evaluation criteria and procedures are discussed in Chapter 5. If evaluation reveals that an alternative is not acceptable, then the Evaluator provides feedback to the Planner in order to formulate constraints for the generation of the next alternative. If no further alternative can be generated, the current description of an organizational setting is changed according to the constraints identified by the Evaluator, and then is analyzed by the Input checker, and so on, iteratively.

Note that the output of the evaluation process needs to be approved by a human designer. **User evaluation interface** presents the selected alternative to the designer together with the summarized evaluation results. Also, it provides the designer with the interface for giving his feedback on why the selected alternative does not satisfy him. On the basis of this feedback the constraints for the generation of the next alternative are formulated and forwarded to the Planner.

The result of this process is a new requirements model, which is, ideally, optimal or, in practice, good enough with respect to all the local and global criteria, and is approved by the designer. Note that after obtaining one satisficable alternative it is possible to repeat the process to generate others, reusing already identified constraints.

Chapter 4

Exploring the Space of Design Alternatives

In this chapter we give the details on how planning is used to support the designer in constructing networks of goal delegations among actors. We explain how the initial organizational setting and the desired plan properties are formalized, and how planning is conducted with the help of an off-the-shelf planning tool.

4.1 Formalizing the input setting and desired properties of a solution

To model an initial organizational settings a designer needs to identify actors and goals, as well as social dependencies among actors. For this purpose, a set of first-order predicates is used, as presented in Table 4.1. The predicates take variables of three types: **actor**, **goal** and **gtype** (goal type) and are classified under the following categories:

Goal properties. To assign types to goals, **type** predicate is used. Goal types are used to group goals into domain specific types in order to allow for specifying the properties of an entire group of goals instead of specifying the same property for each goal separately. Goal refinements are represented using **and/or_subgoal_n** predicates. The temporal order in which goals are achieved is constrained by the **order** predicate. To represent a goal conflict, i.e. the situation in which only one of several goals can be achieved, the **conflict** predicate is used. This predicate is symmetric, that is

$$\forall g_1, g_2 : \text{goal_conflict}(g_1, g_2) \rightarrow \text{conflict}(g_2, g_1)$$

The means-end relation between goals is represented by the **means_end** predicate, which reflects the possibility that the satisfaction of an end goal is accomplished through the satisfaction of the corresponding means goal. When a goal is satisfied, the **satisfied** predicate becomes true.

Actor properties. Actor capabilities are described with **can_satisfy** and **can_satisfy_gt** predicates meaning that an actor has enough capabilities to satisfy either a specific goal,

Goal Properties
<code>type(g : goal, gt : gtype)</code> <code>and_subgoal_n(g : goal, g₁ : goal, ..., g_n : goal)</code> <code>or_subgoal_n(g : goal, g₁ : goal, ..., g_n : goal)</code> <code>order(g₁ : goal, g₂ : goal)</code> <code>conflict(g₁ : goal, g₂ : goal)</code> <code>means_end(g₁ : goal, g₂ : goal)</code> <code>satisfied(g : goal)</code>
Actor Properties
<code>can_satisfy(a : actor, g : goal)</code> <code>can_satisfy_gt(a : actor, gt : gtype)</code> <code>wants(a : actor, g : goal)</code>
Actor Relations
<code>can_depend_on(a₁ : actor, a₂ : actor)</code> <code>can_depend_on_gt(a₁ : actor, a₂ : actor, gt : gtype)</code> <code>can_depend_on_g(a₁ : actor, a₂ : actor, g : goal)</code>

Table 4.1: Formalizing an organizational setting: predicates

or any goal of a specific type. We define the following axiom for `can_satisfy_gt` predicate.

$$\forall a : \text{actor}, gt : \text{gtype}, g : \text{goal} \\ \text{can_satisfy_gt}(a, gt) \wedge \text{type}(g, gt) \rightarrow \text{can_satisfy}(a, g)$$

Initial actor desires are represented with `wants` predicate.

Actor relations. Dependencies among actors are reflected by `can_depend_on`, `can_depend_on_gt`, and `can_depend_on_g` predicates, which mean that one actor can delegate to another actor the fulfilment of any goal, or any goal of a specific type, or a specific goal, respectively. We define the following axioms for `can_depend_on` and `can_depend_on_gt` predicates.

$$\forall a_1, a_2 : \text{actor}, gt : \text{gtype} \\ \text{can_depend_on}(a_1, a_2) \rightarrow \text{can_depend_on_gt}(a_1, a_2, gt), \\ \forall a_1, a_2 : \text{actor}, gt : \text{gtype}, g : \text{goal} \\ \text{can_depend_on_gt}(a_1, a_2, gt) \wedge \text{type}(g, gt) \rightarrow \\ \text{can_depend_on_g}(a_1, a_2, g).$$

Figure 4.1 presents an example of a formalized organizational setting corresponding to the scenario shown in Figure 3.1 (excluding information on roles and is-part-of relations).

4.2 Planning approach

As indicated earlier, we adopt the view of i^* [?] and Tropos [24], where the requirements to socio-technical systems are conceived as networks of delegations among actors. Every delegation involves two actors, where one actor delegates to the other the fulfilment of a goal. The delegatee can either fulfill the delegated goal, or further delegate it, thus creating another delegation relation in the network. Intuitively, these can be seen as *actions* that the designer ascribes to the members of the organization and the system-to-be. Further, the task of constructing such networks can be framed as a *planning problem*

```

TBanking – gtype
Customer Bureau Bank Manager
  JuniorClerk SeniorClerk – actor
BuyHouse FindHouse FindMoney CloseSale GetLoan
  FindLoanPlan ApplyForLoan GetLoanApproved
  GetLoanPlanInfo SelectLoanPlan SendApplication
  GetApplAccepted CheckExtR – goal

(type GetLoanPlanInfo TBanking)
(type GetApplAccepted TBanking)
(type GetLoanApproved TBanking)
(can_depend_on_gt Customer Bank TBanking)
(can_depend_on Bank Manager)
(can_depend_on Bank Bureau)
(can_satisfy Customer FindHouse)
(can_satisfy Customer SelectLoanPlan)
(can_satisfy Customer SendApplication)
(can_satisfy Customer CloseSale)
(can_satisfy Bureau CheckExtR)
(and_subgoal3 BuyHouse FindHouse FindMoney CloseSale)
(and_subgoal3 GetLoan
  FindLoanPlan ApplyForLoan GetLoanApproved)
(and_subgoal2 FindLoanPlan
  GetLoanPlanInfo SelectLoanPlan)
(and_subgoal2
  ApplyForLoan SendApplication GetApplAccepted)
(means_end GetLoan FindMoney)
(wants Customer BuyHouse)

```

Figure 4.1: An extract of formalization of the diagram in Figure 3.1

where selecting a suitable socio-technical system structure corresponds to selecting a plan that satisfies the goals of human, organizational and software actors.

Thus, we have chosen an AI planning approach [122] to support the designer in the process of selecting the best alternative socio-technical structure. Therefore, we need to choose a specification language to represent the planning domain, that is, the initial and the desired states of the world and the actions that can be performed. Then, the solution to the planning problem will be a sequence of actions that makes it possible to reach a desired state of the world starting from the initial state.

The set of predicates introduced in Table 4.1 is used to represent the initial state of an organizational setting. The desired state (or *goal of the planning problem*) is described through the conjunction of **satisfied** predicates: for each **wants(a,g)**, **satisfied(g)** is added to the goal of the planning problem.

A plan, which is constructed to fulfill the goals of system actors, comprises the following actions:

- **Goal satisfaction.** Satisfaction of a goal is an essential action, which can be performed only by an actor who is capable of achieving the goal. The result of this action is the fulfillment of the goal. Action **SATISFIES**($a : actor, g : goal$) represents the fact that goal g is achieved by actor a .
- **Goal delegation.** An actor may choose to delegate one of his goals to another actor. We represent this transfer of responsibilities through action **DELEGATES**($a_1, a_2 : actor, g : goal$). It is performed only if the delegator wants the goal to be fulfilled and trusts that the delegatee will achieve it (i.e. he can actually depend on the delegatee). After having delegated the goal, the delegator is no longer responsible for its fulfillment, and does not care how the delegatee satisfies the goal (e.g. by his own capabilities or by further delegation).
- **Goal decomposition.** Two types of goal refinement are supported by the framework: AND- and OR-decomposition. A way/ways in which a goal can be decomposed is predefined and known to all the system actors. Actions **AND_DECOMPOSES**($a : actor, g, g_1, \dots, g_n : goal$) and **OR_DECOMPOSES**($a : actor, g, g_1, \dots, g_n : goal$) represent the fact that goal g is decomposed into n AND/OR-subgoals by actor a .

Actions are described in terms of preconditions and effects, both being conjunctions of formulas containing the predicates we have introduced, their negations, disjunctions, and universal and existential quantifiers. If a precondition of an action is true in the current state of the world, then the action can be performed; as a consequence of an action, a new state is reached where the effect of the action is true. Figure 4.2 presents a sample sequence of plan actions corresponding to the diagram in Figure 3.1.

4.3 Desired properties of a solution

In this section, we discuss the properties a solution to a planning problem should satisfy. Later, we will define formally the four plan actions introduced above, so that these properties are satisfied by construction.

AND_DECOMPOSES	Customer	BuyHouse	FindHouse	FindMoney	CloseSale
AND_DECOMPOSES	Customer	GetLoan	FindLoanPlan	ApplyForLoan	GetLoanApproved
AND_DECOMPOSES	Customer	FindLoanPlan	GetLoanPlanInfo	SelectLoanPlan	
DELEGATES	Customer	Bank	GetLoanPlanInfo		
SATISFIES	Customer	SelectLoanPlan			
AND_DECOMPOSES	Customer	ApplyForLoan	SendApplication	GetApplAccepted	
SATISFIES	Customer	SendApplication			
DELEGATES	Customer	Bank	GetApplAccepted		
DELEGATES	Customer	Bank	GetLoanApproved		

Figure 4.2: A fragment of a plan corresponding to Figure 3.1

Let $I = (D, F)$ be the initial state of an organizational setting, where D is the domain, i.e. a set of objects (actors, goals and goal types), and F is a set of fluents, i.e. positive or negative grounded literals (predicates introduced in Table 4.1). We refer to the domain and fluent components of a state I as I_D and I_F , respectively. We also assume that

$$predicate(arg_1, \dots, arg_n) \in I_F \rightarrow \forall i arg_i \in I_D,$$

that is, if some predicate is in I_F then all the objects it takes as arguments are in I_D .

Let P be a plan, i.e. a (partially) ordered sequence of actions, G be a set of predicates which characterize the goal of a planning problem. Then, the execution of plan P in state I will result in a new state R of a socio-technical system,

$$R = result(P, I).$$

In this setting, a planning problem consists in constructing such a P that

$$G \subset R_F.$$

This property, referred to as **basic plan property** from now on, in our setting means that all goals of all actors are satisfied after a plan is executed. The property is formalized as shown in line 1 in Table 4.2. In this and further properties, the names of *actions* of a plan P are capitalized, while the names of *predicates* contain lowercase letters only.

A set of important plan properties, referred to as **plan compliance with the initial organizational setting**, consists in ensuring that a plan can actually be executed starting from the initial state of a socio-technical system. Namely, the following properties should hold.

1. Only an actor who has a capability to satisfy a goal is assigned the obligation to satisfy it.
2. A goal can be delegated from one actor to the other, only if the former can actually depend on the latter.
3. A goal can be AND/OR-decomposed only in a predefined way, that is, according to an AND/OR decomposition tree that is given as part of the problem statement.

For the formal representation of these properties see lines 2.1 – 2.3 in Table 4.2.

Another related set of plan properties, referred to as **plan compliance with goal relations**, represent the fact that executing a plan cannot result in violating temporal, conflict, or means-end relations between goals.

To formalize these properties, let us define a *partial order* of plan actions¹. Let a time step number $T(p) \in \{0, N - 1\}$ be associated with each action $p \in P$ where $N \leq |P|$. Namely, p is performed at $T(p)$, which means that p 's preconditions are true at $T(p)$, while the effects become true at $T(p) + 1$. Also let $S(t), t \in \{0, N - 1\}$ be the state of an organizational setting after all the preceding actions $p, T(p) < t$ are performed.

For a plan to be compliant with goal relations, the following properties should hold.

1. Goals should be satisfied in the correct order.
2. Only one of the conflicting goals can be satisfied in a plan.
3. If a means goal is satisfied and there are no satisfied goals which are in conflict with the end goal, then the end goal should become satisfied.

For the formal representation of these properties see lines 3.1 – 3.3 in Table 4.2.

The last group of properties is concerned with the optimality of a plan, meaning that a plan should be such that the goals of system actors are satisfied in a minimal number of steps. In this respect, the following two properties, referred to as **non-redundancy properties**, should hold.

1. A goal cannot be satisfied more than once.
2. Plans are assumed to be minimal in the sense that any subset of the actions of a plan does not satisfy the goal for which the plan was intended.

For the formal representation of these properties see lines 4.1 – 4.2 in Table 4.2. Note that the axiom 4.1 in Table 4.2 states that the same goal cannot be satisfied by different actors, while the fact that the same goal cannot be satisfied twice by the same actor is implicit in that P is a set, and thus, cannot contain duplicates. Also note that the second property implies the absence of delegation loops and unnecessary actions, meaning that no action over the same goal is performed twice by two different actors. Even though the first property can be inferred from the second one, we state both explicitly for the sake of clarity.

Now our aim is to define the plan actions in terms of their preconditions and effects in the way that the resulting plan satisfies all of the above four groups of properties.

4.4 Planning formalization

As we mentioned already, planning actions are described in terms of preconditions and effects, which in turn, are expressed in terms of predicates presented in Table 4.1. After the application of each subsequent plan action, the state of the world changes, namely,

¹Most of the planning algorithms return a (partially) ordered sequence of actions as a resulting plan.

Basic plan property
1. $\forall a : \text{actor}, g : \text{goal}. \text{wants}(a, g) \in I_F \rightarrow \text{satisfied}(g) \in R_F$
Compliance with initial organizational setting
2.1 $\forall a : \text{actor}, g : \text{goal}. \text{SATISFIES}(a, g) \in P \rightarrow \text{can_satisfy}(a, g) \in I_F$
2.2 $\forall a_1, a_2 : \text{actor}, g : \text{goal}. \text{DELEGATES}(a_1, a_2, g) \in P \rightarrow \text{can_depend_on_g}(a_1, a_2, g) \in I_F$
2.3.1 $\forall a : \text{actor}, g, g_1, \dots, g_n : \text{goal}. \text{AND_DECOMPOSES}(a, g, g_1, \dots, g_n) \in P \rightarrow \text{and_subgoal}_n(g, g_1, \dots, g_n) \in I_F$
2.3.2 $\forall a : \text{actor}, g, g_1, \dots, g_n : \text{goal}. \text{OR_DECOMPOSES}(a, g, g_1, \dots, g_n) \in P \rightarrow \text{or_subgoal}_n(g, g_1, \dots, g_n) \in I_F$
Compliance with goal relations
3.1 $\forall a : \text{actor}, g_1, g_2 : \text{goal}. \text{order}(g_1, g_2) \in I_F \rightarrow \text{satisfied}(g_1) \in S_F(\text{T}(\text{SATISFIES}(a, g_2)))$
3.2 $\forall g_1, g_2 : \text{goal}. \text{conflict}(g_1, g_2) \in I_F \rightarrow \neg(\text{satisfied}(g_1) \in R_F \wedge \text{satisfied}(g_2) \in R_F)$
3.3 $\forall p \in P, g_1, g_2 : \text{goal}. \text{means_end}(g_1, g_2) \in I_F \wedge \text{satisfied}(g_1) \in \text{T}(p) \wedge \neg(\exists g' : \text{goal}. \text{satisfied}(g') \in \text{T}(p) \wedge \text{conflict}(g_2, g') \in I_F) \rightarrow \text{satisfied}(g_2) \in \text{T}(p)$
Non-redundancy properties
4.1 $\forall a : \text{actor}, g : \text{goal}. \text{SATISFIES}(a, g) \in P \rightarrow \neg \exists a' \neq a : \text{actor}. \text{SATISFIES}(a', g) \in P$
4.2 $\neg \exists p. ((R' = \text{result}(\{P \setminus p\}, I)) \wedge (G \subset R'_F))$

Table 4.2: Desired plan properties

some of the fluents which describe the state of the target socio-technical system become true, while the others become false.

A plan which satisfies all actor goals, should step by step change the initial state into the final state where **satisfied**(*g*) is true if initially **wants**(*a*, *g*) was true for some actor *a*. Thus, the key idea is to propagate **wants**(*...*, *g*) along the delegation links from the actor who wants *g* initially, towards the actor(s) who are responsible for satisfying either *g* or its subgoals. With this idea in mind, we define the actions as presented in Table 4.3 and explained in the following.

- **SATISFIES**(*a* : actor, *g* : goal). The preconditions of the satisfaction action are the following: *a* wants *g* to be satisfied, *a* is capable of achieving *g* (or any goal of the same type), all the goals that should be satisfied before *g* are satisfied, no goals which are in conflict with *g* are satisfied. The two effects are: *g* is satisfied, *a* no longer wants *g* to be satisfied. Thus, each satisfaction action removes (makes false) one **wants** predicate from the current state of the world.
- **AND_DECOMPOSES**(*a* : actor, *g*, *g*₁, ..., *g*_{*n*} : goal) and **OR_DECOMPOSES**(*a* : actor, *g*, *g*₁, ..., *g*_{*n*} : goal). The preconditions of a decomposition action are: *a* wants *g* to be satisfied, it is possible to AND/OR-decompose *g* into subgoals *g*₁, ..., *g*_{*n*}. As an effect, *a* no longer wants *g* to be satisfied, while *g*₁, ..., *g*_{*n*} are added to the list of *a*'s desires, i.e. *a* now wants *g*₁, ..., *g*_{*n*} to be satisfied. Thus, a decomposition action removes **wants** predicate for the parent goal from the state of the world, but adds (makes true) *n* **wants** predicates, one for each of the subgoals.

SATISFIES ($a : actor, g : goal$)
precondition: $wants(a, g) \wedge can_satisfy(a, g) \wedge$ $\forall g_{prev} : goal \neg(order(g_{prev}, g) \wedge satisfied(g_{prev})) \wedge$ $\forall g' : goal \neg(conflict(g, g') \wedge satisfied(g'))$
effect: $satisfied(g) \wedge \neg wants(a, g)$
AND/OR_DECOMPOSES ($a : actor, g, g_1, \dots, g_n : goal$)
precondition: $wants(a, g) \wedge and/or_subgoal_n(g, g_1, \dots, g_n)$
effect: $\neg wants(a, g) \wedge wants(a, g_1) \wedge \dots \wedge wants(a, g_n)$
DELEGATES ($a_1, a_2 : actor, g : goal$)
precondition: $wants(a_1, g) \wedge can_depend_on_g(a_1, a_2, g)$
effect: $\neg wants(a_1, g) \wedge wants(a_2, g)$

Table 4.3: Actions: preconditions and effects

- **DELEGATES**($a_1, a_2 : actor, g : goal$). The preconditions of a delegation action are: a_1 wants g to be satisfied, a can depend on a_2 either for any goal or for the goal of the type which g belongs to. The effects are the following two: a_1 no longer wants g to be satisfied, while g is added to the list of a_2 's desires, i.e. a_2 now wants g to be satisfied. Thus, a delegation action removes one **wants** predicate from the current state of the world, and at the same time adds one: in a sense, it “passes” **wants** for g from the delegator to the delegatee.

In addition to the above, the following rules should apply:

- When all AND-subgoals or at least one of the OR-subgoals of a goal are satisfied, then **satisfied** should become true for this goal.
- When a means goal of a **means_end** relation is satisfied and there are no satisfied goals which are in conflict with the end goal, then the end goal should become satisfied.

To address these one can define *axioms*, or *derived predicates* that hold in every state of the system and are used to complete the description of the current state [46]. However, due to performance problems introduced by derived predicates in practice, we postpone the discussion of this issue until Section 4.5.

Now, after the planning domain is fully defined, we need to show that **the desired plan properties** (see Table 4.2) **are guaranteed** by our approach.

Basic plan property. Following AI planning algorithms, a plan is constructed in such a way that the formula which represents the goal of the planning problem becomes true after the plan is applied to the initial setting. Thus, all **satisfied(g)** predicates which comprise the goal of the planning problem are true after all the plan actions are executed.

Plan compliance with the initial organizational setting. According to the definitions of the plan actions (see Table 4.3):

- *satisfaction* action cannot be performed by an actor who has no capability to satisfy a goal;
- *delegation* action cannot be performed if the delegator cannot depend on the delegatee;
- *decomposition* of a goal can only be done in accordance with the AND/OR goal tree given for the goal.

Plan compliance with the goal relations. According to the definition of the plan actions (see Table 4.3) and domain axioms:

- a goal cannot be satisfied earlier than any goal it is in *order* with;
- a goal cannot be satisfied if another goal with which the former is in *conflict* is satisfied;
- satisfaction of a *means* goal implies the satisfaction of the corresponding *end* goal.

Regarding the satisfaction of **non-redundancy properties** there are two following observations. Firstly, for the absence of redundancy we rely not on the way our domain is defined, but on the planning approach itself. Basically, as most planning algorithms search for a (locally) optimal plan [122], a plan containing redundant actions can be immediately replaced by a better one (which is shorter and still satisfies the planning problem goal) just by removing a number of actions. Secondly, the property stating that each goal should be satisfied only once, is supported by the fact that once a goal is satisfied, *satisfied* predicate becomes true for it and it cannot be made false by any other action.

4.5 Implementing the planning domain

...

4.5.1 Choosing the planner

An important problem we have faced during the implementation of our approach, is the problem of choosing the “right planner” among off-the-shelf tools available. In the last years many planners have been proposed [101], which are based on different (classes of) algorithms, use different domain representation languages, adopt different heuristics for making the plan search efficient, etc. We have compared a number of planners (see [29] for the details) with respect to following requirements:

- As discussed earlier in this section, we do not want the planner to produce redundant plans. A plan is non-redundant if after deleting an arbitrary action, the resulting plan is no more valid (i.e. it does not allow reaching the desired state of the world from the initial state). Some planners we have tested (e.g. $DLV^{\mathcal{K}}$ [102]), do not satisfy this requirement.

Table 4.4: Compared planners

Planner	Release	URL
DLV ^K	2005-02-23	http://www.dbai.tuwien.ac.at/proj/dlv/K/
IPP 4.1	2000-01-05	http://www.informatik.uni-freiburg.de/koehler/ipp.html
CPT 1.0	2004-11-10	http://www.cril.univ-artois.fr/vidal/cpt.en.html
SGPLAN	2004-06	http://manip.crhc.uiuc.edu/programs/SGPlan/index.html
SATPLAN	2004-10-19	http://www.cs.washington.edu/homes/kautz/satplan/
LPG-td	2004-06	http://zeus.ing.unibs.it/lpg/

- The planner should use PDDL (Planning Domain Definition Language) since it has become a “standard” planning language and many research groups work on its implementation. Moreover, the language should support a number of advanced features that are essential for implementing our planning domain (e.g. negation in a planning problem goal, which is not allowed in IPP [82]). Ideally, the planner should support the last stable version of PDDL, so PDDL3 [62] or at least PDDL 2.2 [46] should be supported.
- It is desirable that the planner is available on both Linux and Windows platforms as a set of the Tropos-based reasoning tools we have developed [109] work on both. However, we understand that most of the available planners are research tools, and so they are often released for the one specific platform only (with Linux being a more frequent choice).

Based on the above requirements, we have chosen LPG-td [90], a fully automated system for solving planning problems, supporting PDDL 2.2 specification language for implementing our planning domain. We do not claim that this is the best or the final choice of a planning tool, as the field of AI planning keeps developing. The use of PDDL as a domain representation language facilitates a lot any future transition from LPG-td to another PDDL-based planner.

4.5.2 Domain preprocessing and PDDL implementation

Now let us discuss two implementation specific questions, namely, the implementation of decomposition actions, and the implementation of domain axioms.

AND/OR-decomposition actions should take $n + 2$ parameters, where n is the number of subgoals of the decomposed goal. As the number of action parameters in PDDL should be fixed, we have to fix an upper bound for n and introduce an `and/or_subgoal_n` predicate and a decomposition action for each $i \leq n$. According to our experience, large values of n is hardly the case in practice. E.g., for all the (medium-size industrial) case studies considered in [110] and [128], n is less or equal than 6. In Figure 4.3(b) an example of PDDL code of the AND-decomposition action for the case of two subgoals is presented.

As discussed in the previous section, **axioms** in our planning domain should be defined for the following cases:

- to infer the satisfaction of a goal from the satisfaction of all its AND-subgoals, or one of its OR-subgoals;
- to infer the satisfaction of an end goal from the satisfaction of its means.

<pre>(: actionSatisfies : parameters(?a – actor ?g – goal) : precondition(and (can_satisfy ?a ?g) (forall(?g1 – goal)(and (or (not(conflict ?g1 ?g)) (not(satisfied ?g1))) (or (not(order ?g1?g)) (satisfied ?g1)) (or (not(means_end ?g1 ?g)) (not(satisfied ?g1))))))) (wants ?a ?g)) : effect(and (satisfied ?g) (not(wants ?a ?g)) (pr_satisfies ?a ?g)))</pre>	<pre>(: actionAND.Decomposes2 : parameters(?a – actor ?g ?g1 ?g2 – goal) : precondition(and (and_subgoal2 ?g ?g1 ?g2) (wants ?a ?g)) : effect(and (wants ?a ?g1) (wants ?a ?g2) (not(wants ?a ?g)) (pr_and_decomposes2 ?a ?g ?g1 ?g2)))</pre>
(a) Satisfaction action	(b) Decomposition action

Figure 4.3: Samples of PDDL code.

Also, as explained in Section ??, the following rules should hold for the planning domain predicates:

- if goal g_1 is in *conflict* with goal g_2 , then g_2 is in conflict g_1 , that is, conflict relations between goals is symmetric;
- predicates defining actors capabilities and possible dependencies either for all goals or for the goals of a specific goal type (`can_depend_on`, `can_depend_on_gt`, `can_satisfy_gt`), should be “instantiated” so that only capabilities and possible dependencies for the concrete goals (`can_depend_on_g` and `can_satisfy`) can be used in action formalization.

However, defining axioms in terms of *derived predicates* [46] increases the complexity of the planning problem to the point where the planner is not efficient anymore. This problem is neither new, nor specific to the planner we have adopted, as there is always a trade-off between the expressiveness and manageability of a formal domain definition language [61]. In a number of works [61, 41, 59] different approaches to the *preprocessing* of complicated planning domains are proposed. The key idea is to define a mapping that transforms a complicated domain to the equivalent one, in which axioms, quantifies, conditional effects and the like, are represented using a restricted (and less expressive) subset of PDDL language. For the domain axioms, the approach consists in introducing new domain actions and modifying the preconditions and effects of the existing actions.

None of the above cited approaches is suitable for dealing with the axioms in our planning domain. The algorithm presented in [61] was proven to be incorrect in [59]. The alternative algorithms presented in [59] and [41] consider only those axioms that cannot affect predicates which are changed in effects of any domain action. However, this is not the case in our domain, as `satisfied` predicate has to be changed both by actions and axioms. Inspired by the above approaches, we propose the following transformation of our planning domain to its equivalent version that does not represent axioms explicitly.

COMBINES_AND ($g, g_1, \dots, g_n : goal$)
precondition: and_subgoal _n (g, g_1, \dots, g_n) \wedge satisfied(g_1) \wedge .. \wedge satisfied(g_n)
effect: satisfied(g)
COMBINES_OR ($g, g_1, \dots, g_n : goal$)
precondition: or_subgoal _n (g, g_1, \dots, g_n) \wedge (satisfied(g_1) \vee .. \vee satisfied(g_n))
effect: satisfied(g)
INFERS ($g_means, g_end : goal$)
precondition: means_end(g_means, g_end) \wedge satisfied(g_means) \wedge $\forall g : goal \neg$ (conflict(g, g_end) \wedge satisfied(g))
effect: satisfied(g_end)

Table 4.5: Additional actions: preconditions and effects

Firstly, we define a number of new planning actions, presented in Table 4.5. **Combines** actions appear in a plan each time the satisfaction of a goal should be inferred from the satisfaction of its subgoals. **Infers** actions support *means-end* relationship between goals. Note that in the preconditions of this action we have to check whether the end goal is in conflict with any of the goals satisfied so far. If so, the satisfaction of the end goal cannot be inferred as, in our framework, the conflict relation is stronger than the means-end relation.

In order to enforce the use of means-end relation and to avoid redundancy, an additional constraint should be added to the precondition of the satisfaction action. As presented in Figure 4.3(a), a goal cannot be satisfied if its satisfaction can be inferred using means-end relation. Note that the meaning of predicate *pr_satisfies* used in Figure 4.3(a), as well as of the other “trace” predicates, will be discussed in Section ??.

The last step concerns preprocessing the planning problem specification. Namely,

- for each conflict(g_1, g_2) predicate in the problem definition, we add (avoiding the duplicates) conflict(g_2, g_1);
- each can_depend_on is replaced by can_depend_on_g predicates for all goals g ;
- each can_depend_on_gt is replaced by can_depend_on_g predicates for all goals g of the corresponding goal type;
- each can_satisfy_gt is replaced by can_satisfy predicates for all goals g of the corresponding goal type.

As a result, with LPG-td, producing a plan for the planning domain with derived predicates takes more than 100 times longer than producing the same plan with the preprocessed domain. This, basically, means that planning with the derived predicates does not scale, and cannot be used in the real-life domains. During the evaluation of a plan, which is discussed in the next section, **Combine** and **Infers** actions are not taken into

considerations, as only those actions which represent a temporal act, rather than just an immediate inference, matter for the evaluation process.

The proposed requirements analysis approach is supported by S&D (Security and Dependability) Tropos Tool [109]. The tool has the interface for the input of actors, goals and their properties. LPG-td is built in the tool, and is used to generate alternative requirements structures, which are then represented graphically using Tropos notation.

MORE ON THE TOOL...

4.5.3 Complexity of the approach – TO FINISH

Complexity observations; heuristics?

4.6 Alternative search techniques – TO FINISH

- Planning with preferences, constraints, numbers, etc.
- Abductive reasoning: B-Tropos and SCIFF [30]
- Datalog
- A* (?)

Chapter 5

A posteriori Evaluation and Feedback Loop

- Properties that cannot be incorporated in the process of automated search: discussion, formalization and examples
 - Cost schemas
 - Criticality
 - Non-functional requirements
- Planning-and-evaluation schema (incl. technical details on how an "improved" solution is generated);
- Human designer in the loop and forms of feedback she can provide;
- General schema of the approach;
- Special cases of feedback loops: class-instance analysis, secure systems, risk minimization.

The alternative social networks generated by the planner need to be evaluated and approved by the designer. However, such an evaluation can be complex enough even for designers with considerable domain expertise, and thus a supporting tool would be beneficial. Alternative networks can be evaluated both from *global* and *local* perspectives, i.e. from the designer's point of view and from the point of view of an individual actor who is part of the network. In this section we discuss a number of evaluation criteria of both perspectives, as well as their application to the requirements networks.

...

5.1 Global evaluation criteria

The optimality of a solution in the global sense could be assessed with respect to the following criteria.

- Length of proposed plan.
- Overall plan cost.
- Degree of satisfaction of non-functional requirements.

5.1.1 Length of proposed plan

The number of actions in the proposed plan is most often the criterium for the planner itself to prefer one solution over another. For those planners which produce partially ordered plans (e.g. IPP [82]), the corresponding measure is the number of time steps in which a plan can be executed. Thus, it can be assumed that a plan produced by a planning tool is already optimal (or, in many cases, locally optimal, e.g. in case of LPG-td) in terms of length minimization.

5.1.2 Overall plan cost

In AI planning, criteria which take costs into consideration, are related to the idea of plan metrics introduced in PDDL 2.1 [52]¹. Plan metrics specify the basis on which a plan is evaluated for a particular problem, and are usually numerical expressions to be minimized or maximized. E.g. a cost or duration can be associated to a domain action, and then a plan is constructed so that the sum of costs/durations of all the plan actions is minimal. However, the complexity of the problem of optimizing a solution with respect to the defined metrics is high, in fact, “the introduction of numeric expressions, even in the constrained way that we have adopted in PDDL 2.1, makes the planning problem undecidable” [52]. Thus, effective use of plan metrics is still a research issue, and, as a consequence, the feature is poorly supported by the available planning tools.

Moreover, the planning metrics are not sufficient in cases where a cost should be assigned to *instances* of a planning action, while the planning metrics in PDDL 2.1 work on a generic rather than an instance level. Namely, we want to assign different costs to `SATISFIES(SeniorClerk,CalcIntRating)` and `SATISFIES(Manager,CalcIntRating)` actions, while planning metrics allow us to assign a cost value only to the “generic” action `SATISFIES(a: actor, g: goal)`. This is not satisfactory, as in our planning domain, costs associated with *satisfying* the same goal are likely to be different for different actors, as well as costs associated with *delegating* the same goal to different actors can differ (e.g. delegating a goal to the colleague sitting next to you is often less costly than delegating the very same goal to the employee of another department who you even do not know in person). Thus, instead of using the planning metrics, we propose to evaluate a plan a posteriori, and then, on the basis of the evaluation results, provide additional constraints for the search for the next, better plan.

¹Another related approach is planning with preferences (see e.g. [12]). However, for reasons similar to those for the planning metrics case, most plan evaluation criteria discussed in this section cannot be incorporated into planning with preferences.

Let us now introduce notions of an action and plan cost, and a general schema we use further in the paper to represent and compare costs.

A cost associated to an instance of a domain action can incorporate a variety of factors, such as the time spent for executing the action, the amount of (different types of) resources used, the associated effort (related to complexity of the action), etc. Let $\vec{c}(p)$, where p is the instance of an action of the form $action_name(arg_1, \dots, arg_n)$, be a cost vector associated to p , where $c_i(p)$ measures the i^{th} cost dimension.

For example, consider the satisfaction of the goal *calculate internal rating*. Let $\vec{c}(\text{SATISFIES}(a, \text{CalcIntRating}))$, where a is the actor who actually satisfies this goal, be of the form $(total_time, doc_support, effort)$. Here the components of the cost vector are (a) the total time spent to satisfy the goal (i.e. to calculate an internal customer rating), (b) the number of instructions and regulations consulted in the bank electronic document base, (c) the subjective complexity that actor a attributes to the action (e.g. characterized as “high”, “medium” or “low”). Then, two concrete examples for *manager* and *senior clerk* actors will be

$$\begin{aligned}\vec{c}(\text{SATISFIES}(\text{Manager}, \text{CalcIntRating})) &= (20', 3, low), \\ \vec{c}(\text{SATISFIES}(\text{SeniorClerk}, \text{CalcIntRating})) &= \\ &= (35', 1, medium).\end{aligned}$$

However, as the entries in a cost vector are, in general, heterogenous and measured in different units, this representation of cost is not satisfactory. Let us then introduce a scaling function $\vec{r}_c(\vec{c}(p))$, which maps heterogenous cost values to natural numbers from 1 to n , with 1 being the lowest and n the highest cost values. In the above example, let $n = 3$, and

$$\begin{aligned}[\vec{r}_c]_1(total_time) &= \begin{cases} 1, total_time \leq 20', \\ 2, 20 < total_time \leq 40', \\ 3, total_time > 40', \end{cases} \\ [\vec{r}_c]_2(doc_support) &= \begin{cases} 1, doc_support \leq 2, \\ 2, 2 < doc_support \leq 5, \\ 3, doc_support > 5, \end{cases} \\ [\vec{r}_c]_3(effort) &= \begin{cases} 1, effort = low, \\ 2, effort = medium, \\ 3, effort = high, \end{cases}\end{aligned}$$

where $[\vec{r}_c]_i(arg_i)$ is a “partial” scaling function for the i^{th} cost factor. In this setting,

$$\begin{aligned}\vec{r}_c(\vec{c}(\text{SATISFIES}(\text{Manager}, \text{CalcIntRating}))) &= (1, 2, 1), \\ \vec{r}_c(\vec{c}(\text{SATISFIES}(\text{SeniorClerk}, \text{CalcIntRating}))) &= (2, 1, 2).\end{aligned}$$

The cost function of a plan P , called also *social cost* in networks literature (see e.g. [95]), is then defined either in the vector form

$$\vec{c}(P) = \sum_{p \in P} \vec{r}_c(\vec{c}(p)),$$

or in the scalar form

$$\bar{c}(P) = \sum_{p \in P} \frac{1}{n} \sum_{i=1}^n [r_c^{\vec{c}}(\bar{c}(p))]_i,$$

where $[\vec{v}(x)]_i$ is the i^{th} element of vector $\vec{v}(x)$.

Note that we do not discuss here neither the way the cost is measured along its various dimensions, nor the way the scaling function is constructed. These tasks are domain specific and require the interference of human experts.

5.1.3 Degree of satisfaction of non-functional requirements

By non-functional requirements (NFR) we mean quality criteria, which allow one to evaluate a socio-technical system, as opposed to functional requirements, which define the behavior of a system [36]. Examples of such criteria are efficiency, reliability, safety, usability, and many others. An important observation is that there can exist many alternative ways to implement system functional requirements, which differ with respect to those non-functional requirements that the designer considers important for a system. The “best” alternative may not exist, as often there are trade-offs between NFR, e.g. performance vs. safety, resilience vs. efficiency, etc. Therefore, the final choice of an appropriate alternative is usually up to a human designer.

To measure the impact of a design alternative on a certain non-functional requirement, either qualitative or quantitative metrics are used. *Qualitative metrics* are relative scales which allow comparing the degree of impact of two design alternatives towards a concrete NFR (e.g. such a metric may allow saying that “alternative X is better than alternative Y in terms of usability”). In goal modelling, qualitative reasoning on non-functional requirements typically means specifying whether the achievement of a goal contributes positively (“+”) or negatively (“-”) to a NFR. In [65] a formal framework for reasoning with goal models is introduced, which distinguishes between full and partial evidence of a goal being either satisfied or denied, and defines the propagation rules of the goal satisfaction labels. A number of works [68, 121, 87] use similar approaches to reason about the choice between functional alternatives on the basis of their contribution to a set of NFR.

In turn, adopting *quantitative metrics* means assigning numerical weights to quantify positive and negative influences of alternative design options on the degree of NFR satisfaction. In some cases, the numerical weights have a domain-specific physical interpretation (e.g. response or recovery times). However, as pointed out in [87], most often it is not clear neither what these numbers mean (as they are subjective), nor where they come from (who is responsible of providing them).

Since the problem of both qualitative and quantitative evaluation metrics in the context of non-functional requirements in software design was studied by many researches (for examples see the references above), and a number of formal frameworks and tools were developed, we do not aim at inventing yet another way of dealing with NFR. A future work direction we consider, is incorporating an existing NFR reasoning technique into our framework.

5.2 Criticality of an actor in a plan

The *criticality* of an actor measures how a social network will be affected in case the actor has been removed from or has left the network. The notion of criticality is tightly connected to that of *resilience* of networks to the removal of their vertices in the social network literature [96]. A significant, while quite obvious related result is that most of the real-life and model networks are highly vulnerable to the removal of their highest-degree vertices, that is, the vertices with the highest number of in- and outgoing links.

In our framework, not only the links between the nodes matter, but also the goals that are assigned to the removed node according to the plan. In the following we discuss both criticality dimensions, that is, the impact of the removal of both satisfaction and delegation actions from a plan². Note that as we consider only non-redundant plans, the removal of actor a *compromises* a plan P in case there exists an action $p \in P$ such that a is one of p 's arguments, $a \in \text{arg}(p)$. Namely, if all actions a participates to are removed from a plan, the remaining actions do not satisfy all goals of the corresponding planning problem.

5.2.1 Leaf goals satisfaction dimension

When an actor is removed from a social network, all the leaf subgoals he was assigned in the plan remain unsatisfied. Namely, a measure of a node's a criticality in plan P is the (weighted) fraction of leaf goals that cannot be satisfied by P after a is removed from the social network constructed in accordance with P . Let an integer number $w(g)$ be the weight of goal g . Basically, $w(g)$ is the measure of importance of g for the socio-technical system defined by a human designer. Then the criticality of actor a in plan P is defined as follows:

$$cr_g(a, P) = \frac{\sum_{\text{SATISFIES}(a, g) \in P} w(g)}{\sum_{\text{SATISFIES}(x, g) \in P} w(g)},$$

where x is an *actor* and g is a *goal*.

If all goals are considered equally important for the system ($\forall g, g' : \text{goal } w(g) = w(g')$), then the criticality can be calculated as follows:

$$cr_g(a, P) = \frac{|g.\text{SATISFIES}(a, g) \in P|}{|(g, x).\text{SATISFIES}(x, g) \in P|},$$

where x is an *actor* and g is a *goal*.

5.2.2 Dependency dimension

Together with an actor, a number of in- and outgoing dependencies for goals are removed from the network. This means that a number of delegation chains become broken and the goals delegated along these chains cannot reach nodes at which they will be satisfied (either directly or after being AND/OR-decomposed). Thus, we can define another set

²By action removal we mean, for instance, temporal unavailability of an actor or the failure of a communication link.

of measures of a node a criticality in plan P , namely, a fraction of “lost” dependencies (ingoing, outgoing, or any type) after a is removed from the social network constructed in accordance with P :

$$\begin{aligned} cr_{in}(a, P) &= \frac{\sum_{\text{DELEGATES}(a', a, g) \in P} w(g)}{\sum_{\text{DELEGATES}(x, y, g) \in P} w(g)}, \\ cr_{out}(a, P) &= \frac{\sum_{\text{DELEGATES}(a, a', g) \in P} w(g)}{\sum_{\text{DELEGATES}(x, y, g) \in P} w(g)}, \\ cr_{dep}(a, P) &= cr_{in}(a, P) + cr_{out}(a, P), \end{aligned}$$

where a' , x , y are *actors* and g is a *goal*.

In a number of other frameworks [?, 60, 25], each dependency link is assigned a certain criticality value, either qualitative or quantitative, which is the measure of how the system will be affected if this dependency fails. In i^* [?] the notion of dependency strength is defined, that is, a dependency is considered to be open (which corresponds to the lowest criticality level), committed, or critical. In [60], in- and outgoing criticality of an actor in a requirements model is defined as the sum of, respectively, the criticality of the in- and outgoing dependencies this actor participates. In addition, the authors present the procedure of complexity and criticality analysis (where complexity is the measure of effort required from an actor to satisfy a specific goal). Namely, the procedure identifies all the actors for which the values of complexity and criticality are greater than the respective predefined maximum values. Then, in order to reduce complexity and criticality of the existing actors, new (software) actors are introduced and the dependencies which violate complexity and criticality constraints are redistributed among these new actors.

In [25] the notion of security criticality is introduced, being the measure of how system security will be affected if the security constraint is violated. An example of a security constraint in our scenario would be to *keep customer's data private* associated to the goal *register customer*. The maximum value of criticality is defined for each actor, and an algorithm is proposed to reduce the criticality (as well as the complexity) of an overloaded actor by redistributing the goals and tasks of this actor to others.

5.2.3 Actor criticality with respect to a set of goals

The proposed criticality measures consider the weighted fraction of leaf goals that are not satisfied or dependencies that are lost after an actor is removed from a network. However, it is also important to quantify the impact of a node removal on the top-level goals of a socio-technical system, or, in general, on any predefined set of non-leaf goals. To address this issue, the following measure is introduced.

Let $G_{aff}(a, P)$ be a set of goals affected by the removal of actor a from the dependency network constructed in accordance with plan P . Namely, if a is removed, all goals in $G_{aff}(a, P)$ cannot be satisfied by the above socio-technical structure. Also, let $G_{dir_aff}(a, P)$ be the set of goals directly affected by the removal of a :

$$\begin{aligned} G_{dir_aff}(a, P) &= \{g : \text{goal. SATISFIES}(a, g) \in P \vee \\ &\quad \exists a' : \text{actor. (DELEGATES}(a', a, g) \in P) \vee \\ &\quad \exists a'' : \text{actor. (DELEGATES}(a, a'', g) \in P)\} \end{aligned}$$

Let G_r be the set of “reference” goals, i.e. top-level or any predefined subset of system goals with respect to which criticality of an actor in a plan will be evaluated. For example, in the e-business case study, G_r can consist of three goals, *get information about loan plans*, *get customer application accepted* and *get loan approved*, or, if one wants to evaluate the criticality of the system with respect to the loan approval process, G_r will consists of its three subgoals, that is, *evaluate loan*, *decide on loan* and *finalize the terms*.

To construct $G_{aff}(a, P)$ goal set corresponding to the set $G_{dir_aff}(a, P)$, the modification of the label propagation algorithm [65] can be used, which allows inferring the (un)satisfiability of top goals by propagating through a goal graph the labels representing evidence for the goals being either satisfied or denied. Satisfiability is propagated bottom-up, from the leaf to the top-level goals, along decomposition, means-end and conflict relations.

The modifications of the algorithm consist in that the propagation starts not from the set of the leaf goals, but from the goals in $G_{dir_aff}(a, P)$ (which are assigned unsatisfiability labels), and those leaf goals which are not the (recursive) AND/OR-subgoals of any goal in $G_{dir_aff}(a, P)$. The unsatisfiability labels of goals $g \in G_{dir_aff}(a, P)$ remain unchained even if their satisfiability is inferred by the label propagation algorithm. The latter may happen if in P goal $g \in G_{dir_aff}(a, P)$ was delegated along the delegation chain which involved the removed actor a , while the satisfaction of g was performed by the actor(s) different from a . The above modifications work due to non-redundancy of the constructed plans, that is, it is assumed that P does not contain alternative ways to satisfy any of the system goals.

After $G_{aff}(a, P)$ goal set is constructed, the criticality of a in P with respect to G_r is defined as follows.

$$cr(a, P, G_r) = \frac{\sum_{g \in G_r \wedge G_{aff}(a, P)} w(g)}{\sum_{g \in G_r} w(g)}.$$

In many cases, the more even the load (in terms of assigned goals and delegations) of system actors, the lower the criticality of each actor. The problem of balancing the load distribution is considered in Section ??, where an evaluation and replanning procedure is introduced. The key idea of this procedure is to formulate the constraints for the construction of the next plan on the basis of the evaluation of the current plan. However, if the workload is balanced, but the subgoals of each goal $g \in G_r$ are distributed among a large fraction of system actors, the criticality of each actor is quite high as the removal of one actor will cause the failure of most of the goals in G_r . One of the approaches to the problem of leveraging high criticality is runtime replanning of those fragment of the social network in which criticality constraints are violated. Though we do not discuss it in this paper, such an approach is a feasible extension of our framework and constitutes another future work direction.

5.3 Local evaluation: game-theoretic insights

A challenging characteristic of requirements analysis for a socio-technical system is the presence of human and organizational actors. These actors can be seen as players in a game theoretic sense as they are self-interested and rational. This may mean that they are

willing to minimize the load imposed personally on them, e.g. they want to constrain the number and the complexity of actions they are involved in³. In a certain sense non-human system actors are players as well as it is undesirable to overload them. Each player has a set of strategies he could choose from, e.g. he could decide to satisfy a goal himself or to pass it further to another system actor. Strategies are based on player capabilities and his relations (e.g. subordination, friendship, or trust, all represented as possible dependencies in our framework) with other human, organizational and software actors in the system.

We assume that each player ascribes a cost to each possible action, as discussed in Section 5.1. Then, it is possible to calculate the cost of a given alternative (or the outcome of the game) for the player by summing up the weights of the plan actions this player is involved in. For each player, minimizing this cost means maximizing the utility of the game. One of the key game theoretical concepts is that of an *equilibrium* [99], which defines the set of strategies, one for each player, which none of independent rational players wants to deviate from. By playing an equilibrium each player maximizes his utility locally, given some constraints. For example, playing the Nash equilibrium means that no player can benefit when deviating from his equilibrium strategy given that all other players play the equilibrium.

However, in non-cooperative setting, there could exist Nash equilibria whose social cost (the sum of individual costs for all players) is much worse than the social cost in the globally optimal situation, called social optimum. To measure the impact of lack of cooperation and coordination on the system effectiveness, the notion of *cost of anarchy* has been introduced in [84]. Cost of anarchy is the ratio between the worst possible Nash equilibrium and the social optimum. There exist numerous studies on the theoretical bounds on the price of anarchy in the specific cases (e.g. [84, 4]), as well as the attempts to design games, i.e. strategies and reward schemas, so that to encourage behaviors close to the social optimum (see e.g. [7, 86], and, more generally, mechanism design theory [40]).

A substantial difficulty in applying game-theoretic ideas to our problem is that all actors of a socio-technical system need to work cooperatively in order to satisfy all initial organizational goals. Differently from classical non-cooperative game theory, where all players choose their strategies independently and simultaneously before the game, in our problem actor choices are closely interrelated. A player cannot independently change his strategy because the new action sequence will very likely be unsatisfactory, i.e. it will not be a solution anymore. So, to satisfy stakeholder goals it is necessary to impose an additional load on some other actors in order to compensate the load the deviating player tries to avoid. The actors on which this additional load is imposed might not be satisfied with the new solution, and will try to deviate from the strategy they were imposed, and so on and so forth. Thus, if one actor wants to deviate from the generated solution, the re-planning is needed to search for another alternative option, which is then evaluated, possibly, to be re-planned again. In the following section, we discuss a “planning-and-evaluation” procedure which aims at finding a good enough (rather than optimal) delegation and assignment socio-technical structure among the available alternatives.

³This is not always the case, as sometimes actors may want the workload to be the maximum they can handle, e.g. in looking for the reward like salary increase or a the recognition of the boss/colleagues.

5.4 Supporting local evaluation

In an ideal setting, the objective of our framework is to produce plans which are optimal with respect to both global and local evaluation criteria. However, choosing the optimum among all available alternatives, which are in the general case exponentially many, might not be feasible in practice. Moreover, as noted by Herbert Simon [112], what makes humans effective (in comparison to machines) is their ability to identify a *satisficing* design as opposed to an optimal one. Thus, our approach to optimization, or, more precisely, to looking for a satisficing solution consists in the following: For all the global and local criteria, thresholds are specified, and a plan that stays within these thresholds is considered to be good enough to be adopted.

In this section we present the revised procedure we devised for optimizing a plan with respect to the local criteria [27]. The problem of evaluating and improving a plan with respect to such global criteria as the overall plan cost and actor criticality, can be faced in an analogous way. We do not report here the details in order not to overload the paper, and focus on optimizing a plan with respect to the local criteria.

An example of a utility function we consider in this paper, is related to workload distribution. We assume that each actor, human, organizational or software, wants to minimize the number and cost of goals he is assigned. The cost of a goal, as it was discussed in Section 5.1, can incorporate a variety of factors, such as the time and effort required to achieve it, resources used, etc. Costs have to be defined explicitly for leaf goals, i.e. for those goals that could be assigned to actors that have capabilities to satisfy them. There is no need to define or calculate a cost for a goal that is to be further decomposed and delegated. Costs are “local” in a sense that the same goal can have different costs for different actors. For each actor there is a maximum complexity (in terms of cost) it can handle, i.e. the sum of costs for all the goals this actor is assigned should be less than a predefined threshold, namely, maximum complexity. If this condition is violated, the actor might be willing to deviate from the imposed assignment.

More precisely, for all actors a_i , $i = \overline{1, n}$ and all goals g_k , $k = \overline{1, m}$, where n and m are the number of actors and goals, respectively, the complexity values are defined:

- $\vec{c}\vec{s}_{ik}$ is the complexity for actor a_i of satisfying goal g_k ;
- $\vec{c}\vec{r}_{ik}$ is the complexity for actor a_i of decomposing goal g_k ;
- $\vec{c}\vec{d}_{ijk}$ is the complexity for actor a_i of delegating goal g_k to actor a_j .

Here we assume that $\vec{c}\vec{s}_{ik}$, $\vec{c}\vec{r}_{ik}$, $\vec{c}\vec{d}_{ijk}$ are of the form $\vec{r}_c(\vec{c}(p))$, $p \in P$, that is, they are “normalized” with the help of scaling function $\vec{r}_c(\cdot)$ (see Section 5.1 for the details).

The cost of a given alternative P for actor a_i is calculated by summing up the costs of actions of P in which a_i is involved, and is denoted by

$$\vec{c}(P, a_i) = \sum_{\text{DELEGATES}(a_i, a_j, g_k) \in P} \vec{c}\vec{d}_{ijk} + \sum_{\text{DECOMPOSES}(a_i, g_k, g_{k1}, \dots, g_{kl}) \in P} \vec{c}\vec{r}_{ik} +$$

$$+ \sum_{\text{SATISFIES}(a_i, g_k) \in P} \vec{c} s_{ik},$$

where $\text{DECOMPOSES}(a_i, g_k, g_{k1}, \dots, g_{kl})$ stands for the AND/OR-decomposition of g_k into l subgoals g_{k1}, \dots, g_{kl} .

After the costs are computed, for each actor the conditions are defined upon which an actor decides whether to deviate from an alternative P or not. The conditions could be either one of the following, or both.

- Actor a_i whose predefined maximum complexity $\vec{c}^{max}(a_i)$ is less than $\vec{c}(P, a_i)$ is willing to deviate from P . This condition is the one we consider in the paper.
- Actor a_i whose predefined upper bound $\vec{c}^{dev\ up}(a_i)$ on cost deviation is less than $\vec{c}(P, a_i) - \text{avg}_i(\vec{c}(P, a_i))$ is willing to deviate from P .

After the costs and maximum complexities are defined, the *evaluation procedure* is organized as follows.

1. Plan P is generated by the planner.
2. Plan cost for each actor is calculated, by summing up the costs of all the action the actor is involved in.
3. Actors willing to deviate from the plan are identified, i.e. actors whose plan cost is greater than the corresponding maximum complexity.
4. One of these actors is selected, namely, actor a_{max} which has the maximum difference δ between plan cost and maximum complexity:

$$\delta(a) = \sum_{i=1}^n [\vec{c}(P, a) - \vec{c}^{max}(a)]_i.$$

5. A subset of actions $P_{dev} \subset P_{a_{max}}$ is formed with the total cost greater or equal to δ , where $P_{a_{max}}$ denotes those actions of P in which a_{max} is involved.
6. The definition of the planning problem is changed in order to avoid the presence of actions of P_{dev} during the next planning iteration.
7. The procedure restarts with the generation of a next plan.

The process stops when a good enough solution is found, i.e. no actors are willing to deviate from it and the designer approves this solution.

Step 6 of the procedure deserves additional explanations. In order to make it possible to avoid the actions contained in P_{dev} in the next plan, we introduce the following “tracing” predicates:

```
pr_satisfies(a : actor, g : goal),
pr_and/or_decomposes(a : actor, g1, g2, ... : goal),
pr_delegates(a1, a2 : actor, g : goal),
```

which become true when the corresponding action takes place. Then, if, for instance, P_{dev} contains a satisfaction action for goal g , the following line is added to the goal of the planning problem: `not pr_satisfies(a_{max} , g)`, which means that the next generated solution cannot contain this action.

Chapter 6

Validation and Scalability of the Approach

- Design time application of the approach: the tool;
- Case studies: on the bases of the tool;
- Scalability (of planning and of the whole approach; of the class-instance approach);

In this chapter, we report on the application of the proposed approach to the e-business case study as well as to a number of other case studies, and then present the results of scalability experiments we have conducted to justify the use of a planning approach for medium size real-life case studies.

6.1 E-business case study

In the early requirements model of the e-business case study in Figure 3.1, the goals *get information about loan plans*, *get application accepted* and *get loan approved* are delegated by the *customer* to the *bank*. To satisfy these goals, the actors representing the bank, i.e. the *manager* and the bank *clerks*, decompose these goal as follows. The first goal, *get information about loan plans*, can be satisfied either via online information request processing, or by allowing a customer to come and ask for the related information in person. The second goal, *get application accepted*, is decomposed into two OR-subgoals, one of which concerns the processing of hand-filled loan applications, while the other refers to the processing of online applications. In both cases, the customer and, after that, her request for a loan are registered. We assume that the bank would like to stick to either online or in-person way of working with customers, that is, conflict relations are defined (a) between goals *provide loan information in person* and *process online loan application*, and (b) between goals *provide loan information on online request* and *process hand-filled application*.

The third goal, *get loan approved*, is decomposed into three AND-subgoals, *evaluate loan*, *decide on loan* and *finalize the contract*. The first subgoal concerns the evaluation of credit worthiness of both new and existing bank customers, and, in the latter case, can be

done either from scratch or using the previous evaluations for the same customer stored in the internal database. In turn, evaluating the customer credit worthiness from scratch, requires the involvement of the credit bureau. The goal of *finalizing the terms* concerns communicating the decision and available options to a customer (in person or via phone call), and, finally, signing the loan contract. Note that there are *order* relations among the goals, e.g. loan evaluation should be done before the final decision can be made and contract signed. We do not report all *order* relations here, though they are part of the planning problem file that is used in the experiments reported below.

All the actors, goals, possible ways of goal decomposition, conflict and order relations are formalized in the problem definition file, which format is the same as in Figure 4.1 and which we do not present here for the space reasons. Also, in this file possible dependencies among actors and actor capabilities are specified. Namely, manager and senior clerk are capable of *calculating internal ratings*, *reviewing existing ratings*, *deciding on loans* and *signing contracts*. Senior and junior clerks are capable of *approving online information requests and applications*, *registering customers and applications* and *preparing contracts*. Manager, senior and junior clerks are capable of *providing information on loan plans in person* and *communicating final decision and contract options to customers by phone or in person*. The goal *check external rating* can be satisfied by the credit bureau. The software system, a new actor introduced to the scenario in the process of requirements analysis, is capable of achieving a number of technological goals, such as *processing online information requests*, *registering customers and online loan requests* and *checking bank database for the existing customer ratings*.

In total, the definition of the planning problem for the e-business case study comprises 52 entities (6 actors, 5 goal types and 41 goals organized in 7 decomposition levels), 91 predicates before and 132 predicates after the preprocessing (see Section 4.5 for the details on the preprocessing of planning problem specification).

Here we illustrate the planning and evaluation procedure presented in Section ???. We assume that only satisfaction actions have non-zero complexity for all the actors, and the complexity of one subgoal is equal to one unit for any actor. Maximum complexities are defined for the *manager* (1 unit) and for the *senior clerk* (3 units). In Table 6.1 the iterations of the planning and evaluation procedure applied to the case study are presented. Five iterations are required to reach a good enough solution. For each iteration we give a short textual description, the costs of the plan constructed on this iteration for each of the actors, and the name of a_{max} actor (see Section ??? for the details), who initiates the deviation from the generated plan as his complexity thresholds are violated. The final alternative is presented in Figure 6.1.

On each iteration, LPG-td took about 5.5 seconds to produce a plan. The plan corresponding to the final alternative consists of 57 actions organized in 33 time steps: LPG-td planner produces partially ordered plans, that is, actions that can be performed in parallel are grouped together.

Let us now discuss the evaluation of the final design alternative with respect to the global criteria presented in Section 5.1.

There can be identified at least two important *non-functional* aspects relevant for the e-business case study. The first aspect concerns *customer satisfaction*, which includes

it#	Description	Actor : Workload	Who deviates
1	Manager answers customer requests on loan plans, provides internal ratings and decides on loans; Junior clerk registers customer applications; Senior clerk communicates final decisions to customers and finalizes contracts.	Manager : 3 Senior Clerk : 3 Junior Clerk : 2 System : 0	Manager
2	Manager decides on loans; Senior clerk provides internal ratings, communicates final decisions to customers and finalizes contracts; customer requests are processed and customer applications are registered automatically by the System and approved by Senior clerk.	Manager : 1 Senior Clerk : 6 Junior Clerk : 0 System : 3	Senior Clerk
3	No solution found; $P_{a_{max}}$ of the previous iteration is revisited and SATISFIES(SeniorClerk, CalcIntRating) is replaced with SATISFIES(SeniorClerk, PrepareContract).	—	—
4	Manager decides on loans; Senior clerk processes customer requests and registers customer applications, provides internal ratings and signs contracts; Junior clerk communicates final decisions to customers and prepares contract templates.	Manager : 1 Senior Clerk : 4 Junior Clerk : 2 System : 0	Senior Clerk
5	Manager decides on loans; Senior clerk provides internal ratings, communicates final decisions to customers and signs contracts; customer requests are processed and customer applications are registered automatically by the System and approved by Junior clerk; Junior clerk prepares contract templates.	Manager : 1 Senior Clerk : 3 Junior Clerk : 3 System : 3	—

Table 6.1: E-business case study: plans and their evaluation

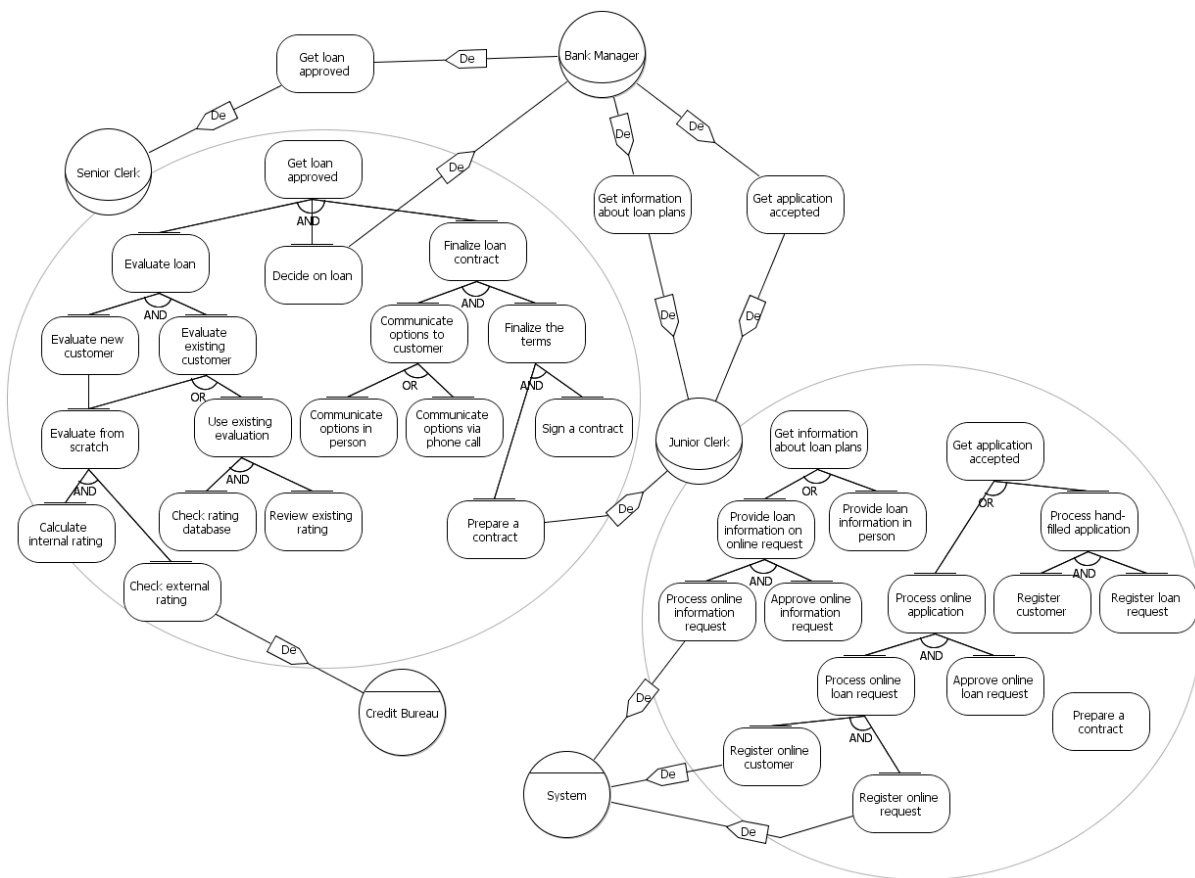


Figure 6.1: E-business case study: adopted solution

quality of service (referred to the direct communication between customers and bank clerks), *usability, or ease of use of the online system, availability* both of clerks and the online system, and *security of the banking system* as perceived by the customer. The second aspect refers to *maximizing bank profits*, which includes *reliability of customer credit trustworthiness assessment, security of the banking procedures, and attractiveness of bank services for the customer*. Both lists are not supposed to be exhaustive, other non-functional aspects can be added after examining the stakeholder interests and the specifics of a concrete bank.

To give an example of comparing two design alternatives with respect to a non-functional requirement, consider the two alternative ways the loan contract terms are communicated to the customer: either in person, or via phone. The customer may consider the former alternative to contribute positively to the security of the procedure just because he perceives personal communication to be more secure for such a private case as the loan contract discussion. Some of the listed non-functional aspect can be addressed only during the further elaboration of the case study. For instance, the questions related to usability and availability of the online banking system are considered during the detailed design of the latter. Some other aspects, such as the availability of the clerks, can be assessed only on the *instantiated* model, in which roles are assigned to concrete agents, and additional instance level constraints are enforced. An example of such constraint, related to security of banking procedures and reliability of customer rating calculations, is the requirement that certain phases of the customer assessment cannot be performed by the same clerks. To a certain extent, this constraint is already addressed at the level of roles, as in the final plan the *manager* takes the final decision on a loan, while the other operations are performed by the *senior clerk*. The problems related to requirements models instantiation and the analysis of the instance level constraints appear to be an interesting research direction and are among our ongoing activities.

The examples of the cost-based global evaluation criteria relevant to the case study are the ones concerned with the resources consumed in each of the alternative configurations of a socio-technical system. For instance, it might be of interest to assess how intensively the printer is used during the loan approval process (the result can be used to adjust the existing infrastructure), or how often the internal document base is consulted during the various stages of the loan approval process (this might be related to the unsatisfactory performance of the bank clerks, or may reveal the insufficient throughput of the internal network). It is relatively straightforward to quantify such criteria and to evaluate the alternatives using the cost-based schema presented in Section 5.1. Note that, as in case of non-functional requirements, some of the cost-based criteria are applicable only at the instance level, for example, in a model containing roles but not concrete agents it does not appear possible to assess the total effort (e.g. in working hours) normally spent for processing one customer application.

The scale of the case study, at least as it is presented in the paper, does not provide the opportunity to fully test the criticality evaluation metrics discussed in Section 5.1. The observations that can be made on the final design alternative presented in Figure 6.1 with respect to criticality are, for instance, the following. The *senior clerk* is a critical actor for the goal *get loan approved*, while the unavailability of the *manager* will not be highly

critical, as the only contribution of the *manager* to this goal is making the final decision. In an alternative system configuration, in which the final decision is taken by the *senior clerk*, the criticality of the *manager* for the goal *get loan approved* is zero, however, as it was discussed above, this contributes negatively to the non-functional requirement for the security and reliability of the loan approval procedure.

6.2 Other case studies - TO FINISH

Medical Information System case study. In [29] we have customized our planning-based framework for the domain of secure system design. The planning domain is defined to guarantee that the resulting socio-technical model satisfies the trust and permission constraints imposed on it (e.g., no goal is delegated along an untrusted link). The framework is applied to the Medical Information System case study, which deals with the payment for medical care. See Section 7.1 for the further details.

Air Traffic Management case study . Another extension of our framework [9] uses risk-based evaluation metrics for selecting a suitable design alternative, and aims at safety critical applications. The approach is evaluated on the basis of Air Traffic Management case study which comes from SERENITY Project [110]. See Section 7.2 for the further details.

E-Voting case study . The illustrative examples used in [28] are taken from the e-voting case study related to the project funded by the Autonomous Province of Trento, which has the goal of providing a smooth transition from the paper-based voting system to new technologies.

Search-and-order MAS case study. See Section 7.3 for the further details.

Software Development Support System case study . Presented in [27].

6.3 Scalability experiments

The case study considered in Section 6.1 allows us to illustrate how the proposed planning-based framework works, and what support it provides to a designer. Still, a reader might wonder what happens in case of much bigger models, i.e. what conclusions can be drawn about the scalability of the approach. In this section, we report on a number of experiments related to the above question.

The main idea of the first part of the experiments was to understand how the growing complexity of a planning problem influences the performance of the approach. We have looked at series of planning problems with (a) growing number of goals to satisfy, and (b) growing complexity of the goal trees. All experiments were conducted using LPG-td planner [90].

A building block of each planning problem file is an *elementary tree* which contains 4 decomposition levels, 15 goals ($G_i, i = \overline{1, 15}$), 2 OR and 4 AND decomposition relations: (or_subgoal2 G1 G2 G3)

N_{trees}	N_{elem}	N_{leafs}	t_{total}
1	15	4	0.14
2	30	8	1.08
3	45	12	5.24
4	60	16	7.43
5	75	20	6.03
6	90	24	15.3
7	105	28	13.25
8	120	32	16.83
9	135	36	19.83
10	150	40	26.52
11	165	44	37.29
12	180	48	ERR

Table 6.2: Experimental results: increasing the number of elementary goal trees

(and_subgoal3 G2 G4 G5 G6)
(or_subgoal2 G4 G9 G10)
(and_subgoal2 G5 G11 G12)
(and_subgoal2 G3 G7 G8)
(and_subgoal3 G7 G13 G14 G15)

All problem files contain 6 actors ($A_i, i = \overline{1,6}$) organized into three levels with respect to the relations between them:

(can_depend_on A1 A2)
(can_depend_on A1 A3)
(can_depend_on A2 A4)
(can_depend_on A2 A5)
(can_depend_on A3 A5)
(can_depend_on A3 A6)

Overlapping capabilities are introduced, namely, each leaf goal can be satisfied by two actors.

In the experiments reported in Table 6.2 the number of elementary goal trees N_{trees} which actor A_1 wants to satisfy was increasing. That is, we studied what happens when the planning problem grows “in breadth”, namely, how the planner behaves in case the number of top goals to be satisfied increases, while the number of levels in goal decomposition trees stays the same. N_{elem} stands for the total number of goals in the planning problem file, N_{leafs} stands for the number of leaf goals satisfied in a plan. t_{total} is the time (in seconds) the planner took to solve the problem, namely, it is the sum of two components, parsing and search time, the latter one being insignificant in all the experiments. *ERR* denotes the situations in which the planner was not able to produce a solution due to the problem complexity. As shown in the table, the planner was able to solve the problems with up to 11 top goals to satisfy (remember that each top goal is an elementary tree containing 15 goals organized into 4 decomposition levels).

In the experiments reported in Table 6.3, the number of goal tree levels N_{lev} for a

N_{lev}	N_{elem}	N_{leafs}	t_{total}	t_{total} with no OR's
4	15	4	0.15	0.14
7	43	7	4.39	4.31
8	51	11	8.88	8.71
9	59	15	6.07	5.48
10	67	19	7.93	7.65
11	75	23	5.94	5.95
12	83	27	12.91	11.84
13	91	31	10.49	8.07
14	99	35	12.58	9.3
15	107	39	12.59	10.48
17	115	43	15.11	10.13
19	123	47	21.61	17
23	131	51	40.39	34.66
24	139	55	ERR	ERR

Table 6.3: Experimental results: increasing the number of goal tree levels

top goal A_1 wants to satisfy was increasing. That is, we study what happens when the planning problem grows “in depth”, namely, how the planner behaves in case the number of top goals (one goal in our case) to be satisfied does not change, while the number of levels in goal decomposition tree, and thus the number of subgoals increases. The meaning of N_{elem} , N_{leafs} , ERR , and t_{total} is the same as in Table 6.2. The first line of numbers in the table refers to the problem file containing one elementary tree, the second line – to the problem file containing three elementary trees, one at levels 1-3, and the other two at levels 4-7. For each of the subsequent lines a problem file was constructed by adding one or two levels of leaf goals to the previous problem file. As shown in the table, the planner was able to solve problems with the top goal having up to 23 decomposition levels.

The last column of values in Table 6.2 refers to t_{total} with no OR's, which is the time the planner took to solve the modified problem, where all OR-decompositions in a problem file were changed to AND-decompositions. The fact that t_{total} with no OR's is less than t_{total} in all the lines is not surprising, as each OR-decomposition doubles the number of alternative solutions, and thus, increases the search space. Additional line of experiments, which is not reported here, revealed that avoiding overlapping capabilities (i.e. reducing a problem file to the one in which each goal can be satisfied by just one actor) did not have a significant impact on t_{total} values.

The results reported in Tables 6.2 and 6.3, in our opinion, justify the scalability of the use of planning in the domain of requirements engineering. According to our experience, requirements models of real-life case studies (including the one considered in this paper) stay within the complexity limits which our planning approach can handle. For instance, for all the case studies reported in [110], the depth of goal decomposition trees is not greater than 10 (while in our approach the planner was able to process the goal tree with depth equal to 23). The number of top goals for the models in [110] is lower than 11, a number of elementary goal trees the planner was able to process in the experiments

reported in Table 6.2.

The second set of experiments, which included not only planning but also the evaluation step, was conducted in order to study the scalability of the whole approach. Based on the obtained results, the following two observations were made. Firstly, parsing and search times were not influenced by additional predicates (negations of tracing predicates, see Section ?? for the details) in the goal of the planning problem. Secondly, convergence to a good enough plan appeared to depend on the ratio of acceptable (in terms of costs) assignments of goals to actors among all possible assignments. For relatively small problems (with 6 actors, 10 leaf goals in a plan, each goal can be satisfied by two actors) the number of iterations towards a good enough plan was less or equal to 10.

Chapter 7

Customizations and Applications of the Approach

...

7.1 Designing secure systems

In this section we present the application of our planning-based approach to the domain of secure and trusted system design. The approach extends Secure Tropos analysis and design framework [66], and is illustrated using a Medical information system case study.¹

The design of secure and trusted software that meets stakeholder needs is an increasingly hot issue in Software Engineering (SE). This quest has led to refined Requirements Engineering (RE) and SE methodologies so that security concerns can be addressed during the early stages of software development (e.g. Secure Tropos vs i^* /Tropos, UMLsec vs UML, etc.). Moreover, industrial software production processes have been tightened to reduce the number of existing bugs in operational software systems through code walk-throughs, security reviews etc. Further, the complexity of present software is such that all methodologies come with tools for automation support.

Secure Tropos methodology [66], being an extension of Tropos [24], is aimed at designing secure systems. Its primitive concepts include those of Tropos and i^* [126], but also concepts that address security concerns, such as ownership, permission and trust. Further, the framework already supports the designer with automated reasoning tools for the verification of requirements as follows:

1. Graphical capture of the requirements for the organization and the system-to-be,
2. Formal verification of the functional and security requirements by
 - completion of the model drawn by the designer with axioms (a process hidden to the designer),
 - checking the model for the satisfaction of formal properties corresponding to specific security or design patterns

¹The material presented in this section was published in [29].

In this framework (as in many other similar RE and SE frameworks) the selection of alternatives is left to the designer. In the following, we show that we can do better.

Indeed, in Secure Tropos requirements are conceived as networks of delegation of execution relations among actors (organizational/human/software agents, positions and roles) for goals, tasks and resources. Every delegation of execution also involves two actors, where one actor depends on the other for the delivery of a resource, the fulfillment of a goal, or the execution of a task. As discussed in Section 4.2, these can be seen as *actions* that the designer ascribes to the members of the organization and the system-to-be. As suggested by Gans et al. [58] the task of designing such networks can then be framed as a planning problem for multi-agent systems: selecting a suitable possible design corresponds to selecting a plan that satisfies the prescribed or described goals of human or system actors. Secure Tropos adds to the picture also the notion of delegation of permission and various notions of trust.

The focus of this section is not on optimal designs: as noted by Herbert Simon [112], what makes humans effective (in comparison to machines) is their ability to identify a satisficing design as opposed to an optimal one. Moreover, we assume that the designer remains in the loop: designs generated by the planner are suggestions to be refined, amended and approved by the designer. The planner is a(nother) support tool intended to facilitate the design process.

7.1.1 Secure Tropos and a motivating case study

Secure Tropos [66] is a RE methodology for modeling and analyzing functional and security requirements, extending the Tropos methodology [24]. This methodology is tailored to describe both the system-to-be and its organizational environment starting with early phases of the system development process. The main advantage of this approach is that one can capture not only the *what* or the *how*, but also the *why* a security mechanism should be included in the system design. In particular, Secure Tropos deals with business-level (as opposed to low-level) security requirements. The focus of such requirements includes, but is not limited to, how to build trust among different partners in a virtual organization and trust management. Although their name does *not* mention security, they are generally regarded as part of the overall security framework.

Secure Tropos uses the concepts of actor, goal, task, resource and social relations for defining entitlements, capabilities and responsibilities of actors. Actors' desires, entitlements, capabilities and responsibilities are defined through social relations. In particular, Secure Tropos supports *requesting*, *ownership*, *provisioning*, *trust*, and *delegation*. Requesting identifies desires of actors. Ownership identifies the legitimate owner of a goal, task and resource, that has full authority on access and disposition of his possessions. Provisioning identifies actors who have the capabilities to achieve a goal, execute a task or deliver a resource. We illustrate the use of the above concepts with the help of the Medical information system (IS) for the payment for medical care case study, presented in [29].

Example 1. The Health Care Authority (HCA) is the “owner” of the goal **provide medical care**; that is, it is the only one that can decide who can provide medical care and through what process. On the other hand, **Patient** wants this goal to be fulfilled. This goal can be

AND decomposed into two sub-goals: provisioning of medical care and payment for medical care. The Healthcare Provider has the capability for the provisioning of medical care, but it should wait for authorization from HCA before doing so.

Delegation of execution is used to model situations where an actor (the delegator) delegates the responsibilities to achieve a goal, execute a task, or deliver a resource to another actor (the delegatee) since he has not the capability to provide one of above by himself. It corresponds to the actual choice of the design. *Trust of execution* represents the belief of an actor (the trustor) that another actor (the trustee) has the capabilities to achieve a goal, execute a task or deliver a resource. Essentially, delegation is an action due to a decision, whereas trust is a mental state driving such decision. Tropos dependency can be defined in terms of trust and delegation [?]. Thus, a Tropos model can be seen as a particular Secure Tropos model. In order to model both functional and security requirements, Secure Tropos introduces also relations involving permission. *Delegation of permission* is used when in the domain of analysis there is a formal passage of authority (e.g. a signed piece of paper, a digital credential, etc.). This relation is used to model scenarios where an actor authorizes another actor to achieve a goal, execute a task, or deliver a resource. It corresponds to the actual choice of the design. *Trust of permission* represents the belief of an actor that another actor will not misuse the goal, task or resource.

Example 2. The HCA must choose between different providers for the welfare management for executives of a public institution. Indeed, since they have a special private-law contract, they can qualify for both the INPDAP and INPDAI² welfare schemes. The INPDAP scheme requires that the Patient partially pays for medical care (with a ticket) and the main cost is directly covered by the HCA. On the contrary, the INPDAI scheme requires that the Patient pays in advance the full cost of medical care and then gets reimbursed. Once an institution has decided the payment scheme, this will be part of the requirements to be passed onto the next stages of system development. Obviously, the choice of the alternative may have significant impacts on other parts of the design.

Figure 7.1 summarizes the above examples in terms of a Secure Tropos model. In this diagram, actors are represented as circles and goals as ovals. Labels **O**, **P** and **R** are used for representing ownership, provisioning and requesting relations, respectively. Finally, we represent trust of permission and trust of execution relationships as edges respectively labelled **Tp** and **Te**.

Once a stage of the *modeling phase* is concluded, Secure Tropos provides mechanisms for the verification of the model [66]. This means that the design process iterates over the following steps:

- model the system;
- translate the model into a set of clauses (this is done automatically);
- verify whether appropriate design or security patterns are satisfied by the model.

Through this process, one can verify the compliance of the model with desirable properties. For example, it can be checked whether the delegator trusts that the delegatee

²INPDAP (Istituto Nazionale di Previdenza per i Dipendenti dell'Amministrazione Pubblica) and INPDAI (Istituto Nazionale di Previdenza per i Dirigenti di Aziende Industriali) are two Italian national welfare institutes.

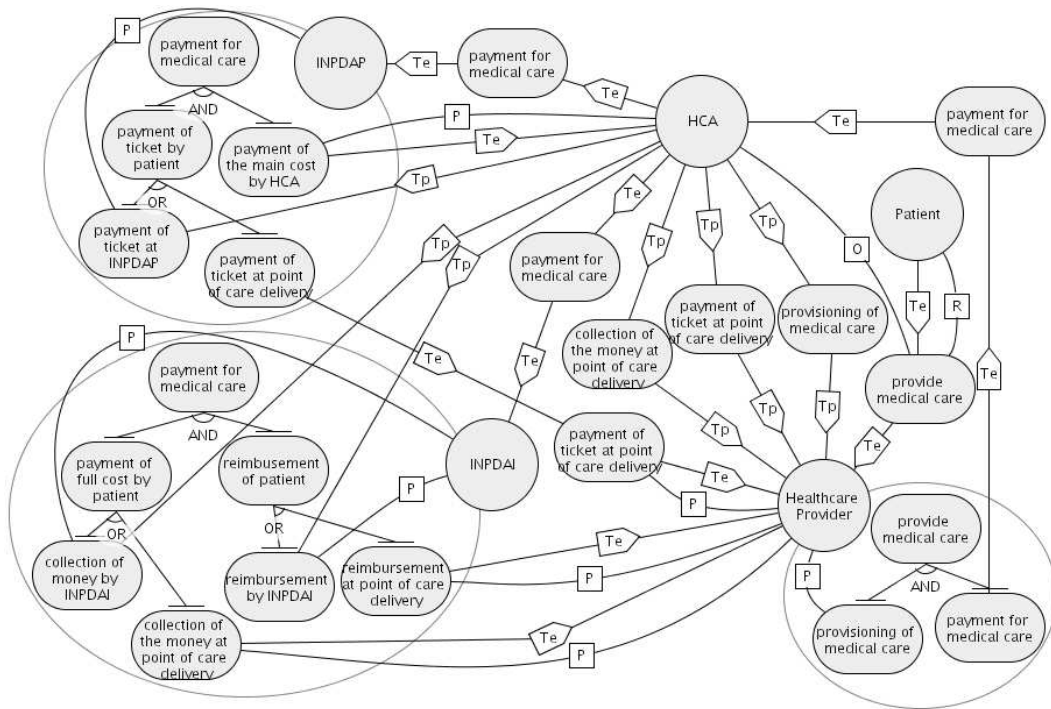


Figure 7.1: Medical IS example: Secure Tropos model

will achieve a goal, execute a task or deliver a resource (trust of execution), or will use a goal, task or resource correctly (trust of permission). Other desirable properties involve verifying whether an actor who requires a service, is confident that it will be delivered. Furthermore, an owner may wish to delegate permissions to an actor only if the latter actually does need the permission. This is done, for example, to avoid the possibility of having alternate paths of permission delegations. Secure Tropos provides the support for identifying all these situations.

However, the “standard” automated reasoning capabilities of Secure Tropos are only able to check that subtle errors are not overlooked. This is rather unsatisfactory from the point of view of the designer. Whereas he may have a good understanding of possible alternatives, he may not be sure which is the most appropriate alternative for the case at hand. This is particularly true for delegations of permission that need to comply with complex privacy regulations (see [?]).

Figures 7.2(a) and 7.2(c) present fragments of Figure 7.1, that point out the potential choices of the design. The requirements engineer has identified trust relations between the HCA and INPDAP and INPDAI. However, when passing the requirements onto the next stage only one alternative has to be selected because that will be the system that is chosen. Figures 7.2(b) and 7.2(d) present the actual choices corresponding to the potential choices presented in Figures 7.2(a) and 7.2(c), respectively.

To support the designer in the process of selecting the best alternative, we proposed to use a modification of the planning-based approach presented in Chapter 4 of the thesis. In the following, we present the modified formalization of the planning domain.

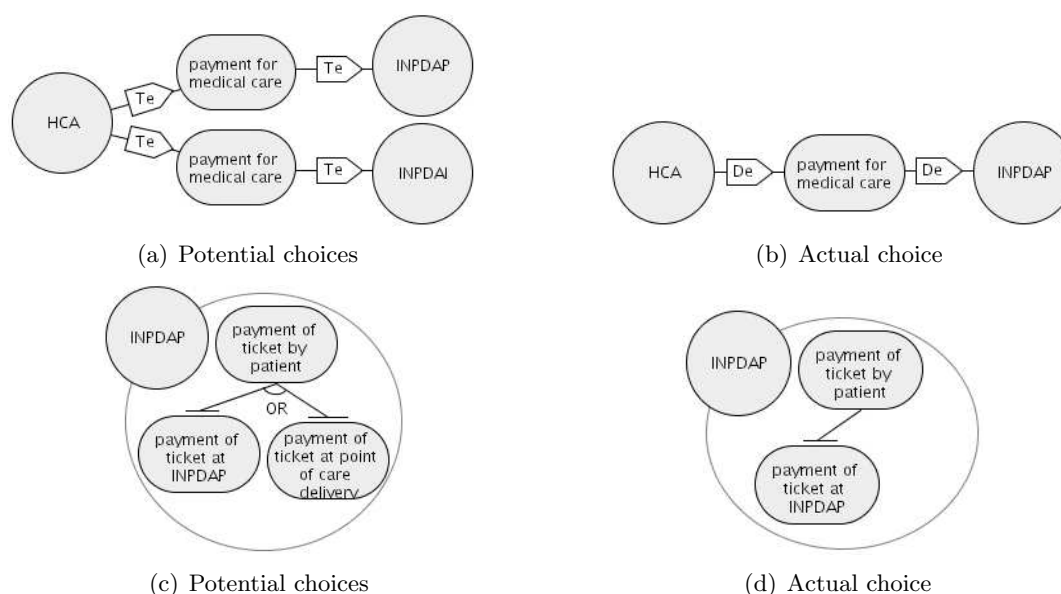


Figure 7.2: Secure systems design: alternatives

Goal Properties
AND_decomposition _n (g : goal, g ₁ : goal, ..., g _n : goal)
OR_decomposition _n (g : goal, g ₁ : goal, ..., g _n : goal)
satisfied(g : goal)
Actor Properties
provides(a : actor, g : goal)
requests(a : actor, g : goal)
owns(a : actor, g : goal)
Actor Relations
trustexe(a : actor, b : actor, g : goal)
trustper(a : actor, b : actor, g : goal)

Table 7.1: Secure systems design: primitive predicates

7.1.2 Planning domain

Table 7.1 presents the predicates used to describe the *initial state of the system* in terms of actor and goal properties, and social relations among actors. We use

- AND/OR_decomposition to describe the possible decomposition of a goal;
- provides, requests and owns to indicate that an actor has the capabilities to achieve a goal, desires the achievement of a goal, and is the legitimate owner of a goal, respectively;
- trustexe and trustper to represent trust of execution and trust of permission relations, respectively.

The desired state of the system (or *goal of the planning problem*) is described through the conjunction of predicates satisfied derived from the requesting relation in the initial state. Essentially, for each request(a,g) we need to derive satisfied(g).

Basic Actions
DelegateExecution(a : actor, b : actor, g : goal)
DelegatePermission(a : actor, b : actor, g : goal)
Satisfy(a : actor, g : goal)
AND_Decompose _n (a : actor, g : goal, g ₁ : goal, . . . , g _n : goal)
OR_Decompose _n (a : actor, g : goal, g ₁ : goal, . . . , g _n : goal)
Absence of Trust
Negotiate(a : actor, b : actor, g : goal)
Contract(a : actor, b : actor, g : goal)
DelegateExecution_under_suspicion(a : actor, b : actor, g : goal)
Fulfill(a : actor, g : goal)
Evaluate(a : actor, g : goal)

Table 7.2: Secure systems design: actions

Actions are listed in Table 7.2, and defined in terms of preconditions and effects as follows:

- **Satisfy.** Following the definition of goal satisfaction given in [66], we say that an actor satisfies a goal only if the actor wants and is able to achieve the goal, and – last but not least – he is entitled to achieve it. The effect of this action is the fulfillment of the goal.
- **DelegateExecution.** This action is equivalent to **goal delegation** action defined in Section 4.2. Namely, in case an actor does not have enough capabilities to achieve assigned goals by himself, he has to delegate their execution to other actors. This is performed only if the delegator requires the fulfillment of the goal and trusts that the delegatee will achieve it.
- **DelegatePermission.** In the initial state of the system, only the owner of a goal is entitled to achieve it. However, this does not mean that he wants it or has the capabilities to achieve it. On the contrary, in the system there may be some actors that want a goal to be achieved and others that can achieve it. Thus, the owner could decide to authorize trusted actors to achieve the goal. The formal passage of authority takes place when the owner issues a certificate that authorizes another actor to achieve the goal. We represent the act of issuing a permission through action **DelegatePermission** which is performed only if the delegator has the permission on the goal and trusts that the delegatee will not misuse the goal. The consequence of this action is to grant rights (on the goal) to the delegatee, which, in turn, can re-delegate them to other trusted actors.
- **AND/OR_Decompose.** These actions are equivalent to **goal AND/OR-decomposition** actions defined in Section 4.2. The effect of **AND_Decompose** and **OR_Decompose** is that the actor who refines the goal focuses on the fulfillment of subgoals instead of the fulfillment of the initial goal. One may argue if decomposing a goal really takes time, and thus, if it is reasonable to treat it as an action. However, a goal may be decomposed in different ways. Thus, we assume that the act of thinking on how it can be decomposed takes time.

The domain *axioms* concerning the propagation of goal satisfaction along goal refinement are defined exactly the same as in Section 4.3. Moreover, axioms are used to derive

and propagate entitlements. Since the owner is entitled to achieve his goals, execute his tasks and access his resources, actors' entitlements should be propagated top-down along goal refinement.

Delegation and contract

Many business and social studies have emphasized the key role played by trust as a necessary condition for ensuring the success of organizations [?]. However, common sense suggests that fully trusted domains are simply idealizations. Actually, many domains require that actors who do not have the capabilities to fulfill their objectives, must delegate the execution of their goals to other actors even if they do not trust the delegates. The presence (or lack) of trust relations among system actors particularly influences the strategies to achieve a goal [?]. In other words, the selection of actions to fulfill a goal changes depending on the belief of the delegator about the possible behavior of the delegatee. In particular, if the delegator trusts the delegatee, the first is confident that the latter will fulfill the goal and so he does not need to verify the actions performed by the delegatee. On the contrary, if the delegator does not trust the delegatee, the first wants some form of control on the behavior of the latter.

Different solutions have been proposed to ensure for the delegator the fulfillment of his objectives. A first batch of solutions comes from transaction cost economics and contract theories that view a contract as a basis for trust [?]. This approach assumes that a delegation must occur only in the presence of trust. This implies that the delegator and the delegatee have to reach an agreement before delegating a service. Essentially, the idea is to use a contract to define precisely what the delegatee should do and so establish trust between the delegator and the delegatee. Other theories propose models where effective performance may occur also in the absence of trust [?]. Essentially, they argue that various control mechanisms can ensure the effective fulfillment of actors's objectives.

In our framework, we propose a solution for delegation of execution that borrows ideas from both approaches. The case for delegation of permission is similar. The process of delegating in the absence of trust is composed of two phases: *establishing trust* and *control*. The establishing trust phase consists of a sequence of actions, namely **Negotiate** and **Contract**. In **Negotiate** the parties negotiate the duties and responsibilities accepted by each party after delegation. The postcondition is an informal agreement representing the initial and informal decision of parties to enter into a partnership. During the execution of **Contract** the parties formalize the agreement established during negotiation. The postcondition of **Contract** is a trust "under suspicion" relation between the delegator and the delegatee. Once the delegator has delegated the goal and the delegatee has fulfilled the goal, the first wants to verify if the latter has really satisfied his objective. This control is performed using action **Evaluation**. Its postcondition is the "real" fulfillment of the goal. To support this solution we have introduced some additional actions (last part of Table 7.2) to distinguish the case in which the delegation is based on trust from the case in which the delegator does not trust the delegatee.

Sometimes establishing new trust relations might be more convenient than extending existing trust relations. A technical "side-effect" of our solution is that it is possible to control the length of trusted delegation chains. Essentially, every action has a unit

cost. Therefore, refining an action into sub-actions corresponds to increasing the cost associated with the action. In particular, refining the delegation action in absence of trust guarantees that the framework first try to delegate to trusted actors, but if the delegation chain results too long it can decide to establish a new trust relation rather than to follow the entire trust chain.

7.1.3 Case study: experimentations

The above described planning domain for the design of secure systems was specified in PDDL 2.2 [46] and tested with LPG-td planner [90].

The planning domain is defined so that the desired plan properties defined in Section 4.3 (the basic plan property, plan compliance with the initial organizational setting and non-redundancy properties) are satisfied by construction. Note that in this case non-redundancy is related to *need-to-know property* of a design decision, which states that the owner of a goal, a task or a resource wants that only the actors who need permission on its possession are authorized to access it. This means that only the actor that achieves a goal, executes a task or delivers a resource, and the actors that belong to the delegation of permission chain from the owner to the provider should be entitled to access this goal, task or resource.

We have applied our approach to the Medical IS presented in Figure 7.1. The desired state of the system is obviously one where the patient gets medical care. The PDDL 2.2 specification of the planning problem is presented in Figure 7.3. Figure 7.4 shows the solution proposed by the planner. The first choice of the planner is one of the two sub-optimal alternatives. Enforcing the planner for further search, we get the optimal solution (i.e., the plan composed of the fewer number of actions than any other plan). It is interesting to see that the planner has first provided a solution with INPDAP, then a solution with INPDAI, and then, finally, a revised solution with INPDAP.

.....

7.2 Using risk analysis to evaluate design alternatives

In this section we present the application of our approach to the domain of agent-based safety critical systems. We show how to incorporate *risk* concerns into the process of a multi-agent system design (and runtime re-design), and illustrate the proposal with the help of an Air Traffic Management case study.³

Multi-agent systems (MAS) have recently proved to be a suitable approach for the development of real-life information systems. The characteristics they exhibit (e.g., autonomy and ability to coordinate), are crucial for safety critical and responsive systems [117]: agents can work independently and respond to the events (e.g., failure, exceptional situation, unexpected traffic, etc.) as quick and correct as needed.

In a safety critical system, human lives depend heavily on the availability and reliability of the system [11]. Therefore, countermeasures are introduced to mitigate the effects of occurring failures (i.e., to cope with the risks that such failures introduce). For

³The material presented in this section was published in [9].

```

: objects
  actor Pat HCA HP INPDAP INPDAI
  goal ProvideMC ProvisioningMC PaymentMC
  goal PaymentTicket PaymentHCA
  goal PaymentTicketHP PaymentTicketINPDAP
  goal PaymentFullCost Reimbursement
  goal CollectionINPDAI CollectionHP
  goal ReimbursementINPDAI ReimbursementHP
: goal
  satisfied ProvideMC
: init
  owns HCAProvideMC
  request Pat ProvideMC
  provide HP ProvisioningMC
  provide HCA PaymentHCA
  provide INPDAP PaymentTicketINPDAP
  provide HP PaymentTicketHP
  provide INPDAI CollectionHCA
  provide INPDAI ReimbursementINPDAI
  provide HP CollectionHP
  provide HP ReimbursementHP
  trustexe Pat HP ProvideMC
  trustper HCA HP ProvisioningMC
  trustexe HP HCA PaymentMC
  trustexe HCA INPDAP PaymentMC
  trustexe HCA INPDAI PaymentMC
  trustexe INPDAP HCA PaymentHCA
  trustper HCA INPDAP PaymentTicketINPDAP
  trustexe INPDAP HP PaymentTicketHP
  trustper HCA HP PaymentTicketHP
  trustper HCA INPDAI CollectionINPDAI
  trustexe INPDAI HP CollectionHP
  trustper HCA HP CollectionHP
  trustper HCA INPDAI ReimbursementINPDAI
  trustexe INPDAI HP ReimbursementHP
  AND_decomposition_2 ProvideMC ProvisioningMC PaymentMC
  AND_decomposition_2 PaymentMC PaymentTicket PaymentHCA
  AND_decomposition_2 PaymentMC PaymentFullCost Reimbursement
  OR_decomposition_2 PaymentTicket PaymentTicketINPDAP PaymentTicketHP
  OR_decomposition_2 PaymentFullCost CollectionINPDAI CollectionHP
  OR_decomposition_2 Reimbursement ReimbursementINPDAI ReimbursementHP

```

Figure 7.3: Medical IS example: the planning problem in PDDL 2.2

instance, OASIS Air Traffic Management system [88] exploits autonomous and responsive behavior of agents to manage airspace and schedule air traffic flow. OASIS monitor components/agents compare the prediction of aircraft locations (i.e., the results of predictor agents) and the actual aircraft positions. In case there is a significant discrepancy, the monitor agent notifies the scheduler agent to re-schedule the landing time of a related aircraft. Therefore, the monitor agent is seen as a countermeasure to prevent the risk of aircraft collision which can occur due to the discrepancy prediction done by the predictors.

Optimal Plan
DelegateExecution Pat HP ProvideMC AND.Decompose HP ProvideMC ProvisioningMC PaymentMC DelegatePermission HCA HP ProvisioningMC Satisfy HP ProvisioningMC DelegateExecution HP HCA PaymentMC DelegateExecution HCA INPDAP PaymentMC AND.Decompose INPDAP PaymentMC PaymentTicket PaymentHCA DelegateExecution HCA INPDAP PaymentHCA Satisfy HCA PaymentHCA OR.Decompose INPDAP PaymentTicket PaymentTicketINPDAP PaymentTicketHP DelegatePermission HCA INPDAP PaymentTicketINPDAP Satisfy INPDAP PaymentTicketINPDAP
Sub-optimal Plan 1
DelegateExecution Pat HP ProvideMC AND.Decompose HP ProvideMC ProvisioningMC PaymentMC DelegatePermission HCA HP ProvisioningMC Satisfy HP ProvisioningMC DelegateExecution HP HCA PaymentMC DelegateExecution HCA INPDAI PaymentMC AND.Decompose INPDAI PaymentMC PaymentFullCost Reimbursement OR.Decompose INPDAI PaymentFullCost CollectionINPDAI CollectionHP DelegatePermission HCA INPDAI CollectionINPDAI Satisfy HCA CollectionINPDAI OR.Decompose INPDAI Reimbursement ReimbursementINPDAI ReimbursementHP DelegatePermission HCA INPDAI ReimbursementINPDAI Satisfy INPDAI ReimbursementINPDAI
Sub-optimal Plan 2
DelegateExecution Pat HP ProvideMC AND.Decompose HP ProvideMC ProvisioningMC PaymentMC DelegatePermission HCA HP ProvisioningMC Satisfy HP ProvisioningMC DelegateExecution HP HCA PaymentMC DelegateExecution HCA INPDAP PaymentMC AND.Decompose INPDAP PaymentMC PaymentTicket PaymentHCA DelegateExecution HCA INPDAP PaymentHCA Satisfy HCA PaymentHCA OR.Decompose INPDAP PaymentTicket PaymentTicketINPDAP PaymentTicketHP DelegateExecution INPDAP HP PaymentTicketHP DelegatePermission HCA HP PaymentTicketHP Satisfy HP PaymentTicketHP

Figure 7.4: Medical IS example: the chosen design alternative

However, since designers never have the complete knowledge about the future, they are not able to define all the necessary countermeasures to prevent any kind of failures/risks. One of the solutions is to enable agents with automatic capabilities to deal with the effect of failures at runtime, as done in ANTS (Autonomous Nano Technology Swarm) mission of NASA [117]. ANTS is comprised of autonomous agents (ruler, messenger and worker agents) designed to cooperate in the process of asteroid exploration. A ruler organizes workers and messengers so that they can explore an asteroid. However, a ruler may need to re-organize the agents at runtime, for example, in case one of the worker agents has been hit by an asteroid. This action is seen as a countermeasure that compensates for

the missing agent in the asteroid exploration mission.

In this section, we present an approach which allows the designer to evaluate alternative system configurations in terms of risk-based metrics either during the initial MAS design, or at a certain time point at runtime when the need occurs to re-design an existing MAS structure in response to the changing outer conditions. The use of risk analysis during the evaluation of alternative system designs helps a designer to ensure the reliability and availability of a safety critical system, as long as an alternative with risk below the predefined thresholds is chosen. Our framework is meant to be a Computer-Aided Software Engineering (CASE) tool that *supports* a designer in defining and risk-based evaluation of MAS design alternatives, but *does not exclude* the designer from the loop.

In the following, we first introduce an Air Traffic Management case study, then detail the approach, and finally present its application to the case study.

7.2.1 A motivating case study

To illustrate our approach, we use the Air Traffic Management (ATM) case study which has been studied in the SERENITY Project⁴. Not surprisingly, an ATM system falls into the category of safety-critical systems as it is closely related to the safety of human lives, and, therefore, it is required to be available and reliable all the time of its operation. However, there can occur many events, not known in advance, which can obstruct the system and compromise its safety. For example, the aircraft traffic in a sector may exceed the safety threshold considered during designing the ATM. Further in this section, we will propose a way to mitigate the risk introduced by a malicious event via the system risk-based evaluation and redesign at runtime so that the new MAS design will compensate the effect of risk. Before presenting the approach, let us describe the case study.

An Air traffic Control Center (ACC) is a body authorized to provide air traffic control (ATC) services in a certain airspace. These services comprise controlling an aircraft, managing the airspace, managing flight data of a controlled aircraft, and providing information on the air situation. Suppose there are two adjacent ACCs, ACC-1 and ACC-2. As can be seen in Figure 7.5, the airspace of ACC-1 is surrounded by the airspace of ACC-2. An ACC organizes its airspace into several adjacent volumes, called sectors. For instance, the airspace of ACC-1 is divided into 2 sectors (Sec 1-1 and Sec 2-1), and the airspace of ACC-2 is divided into 4 sectors (Sec 1-2, 2-2, 3-2, and 4-2). Each sector is operated by a team, consisting of a controller (Sec 1-1 has C1-1 as a controller), and a planner (P1-1 is a planner for Sec 1-1). For the ease of communication, several adjacent sectors in an ACC are supervised by a supervisor (SU1-1 supervises Sec 1-1 and 2-1 and SU1-2 supervises Sec 1-2 and 2-2). In this scenario, a supervisor is a human agent, while controllers and planners are software agents. To simplify, we call both human agent and software agent actors. The supervisor may also play the role of a designer who approves/declines the new plan and, consequently, is responsible for all the effects of this decision.

The scenario starts from the normal operation of ATM in which SU1-1 delegates control of sector 1-1 to team 1, which consists of C1-1 and P1-1. C1-1 and P1-1 work together providing ATC services to aircrafts in sector 1-1. C1-1 controls aircrafts to guarantee the

⁴<http://www.serenity-project.org>

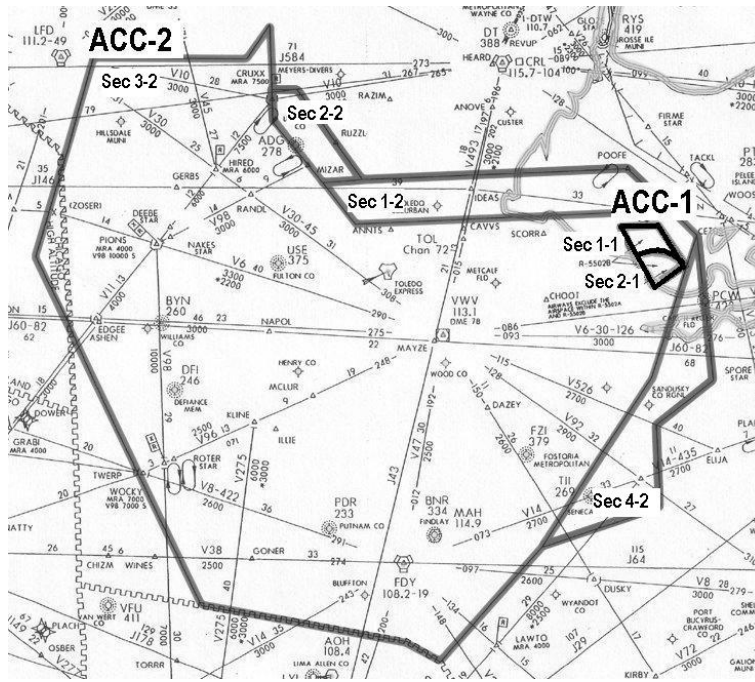


Figure 7.5: ATM case study: air space division between ACC-1 and ACC-2

safe separation of each aircraft vertically and horizontally, and P1-1 manages the flight data of the controlled aircrafts and the airspace of sector 1-1, thus supporting C1-1 in controlling activities.

Now let us imagine that one day during summer holidays a flight bulletin appears that states that there is going to be an increase in the en-route traffic to sector 1-1. According to the analysis made by P1-1, this will go beyond the capacity that can be safely handled by a controller (C1-1). Thus, SU1-1 needs to redesign his sectors so that the en-route traffic can be handled safely, which can be done by:

- dividing his airspace into smaller sectors so that each controller covers a smaller area and, consequently, the number of sectors that are supervised by SU1-1 is increased;
- delegating a part of the airspace to the adjacent supervisor (from the same or different ACC).

Each alternative introduces different requirements. For instance, when dividing the airspace, SU1-1 needs to ensure the availability of a controlling team (G_{14}, G_{21}) and the availability of a set of CWP⁵. (G_{15}, G_{22}). Conversely, if SU1-1 decides to delegate a part of his airspace to another supervisor, then SU1-1 needs to define delegation schema (G_{10}) and to have sufficient level of “trust” towards the target supervisor and his team to manage the delegated airspace. Moreover, SU1-1 needs to be sure that the target supervisor has sufficient infrastructure (e.g., radars, radio communication coverage) to provide ATC services in the delegated airspace.

⁵Controller Working Position (CWP) is a set of resources allocated to support a controller to perform his tasks

Goal Properties
type(g : goal)
AND_decomposition _n (g : goal, g ₁ : goal, . . . , g _n : goal)
OR_decomposition _n (g : goal, g ₁ : goal, . . . , g _n : goal)
satisfied(g : goal)
criticality_h/m/l(g : goal)
criticality_gt_h/m/l(gt : gtype)
Actor Properties
can_satisfy(a : actor, g : goal)
can_satisfy_gt(a : actor, g : gtype)
wants(a : actor, g : goal)
Actor Relations
can_depend_on_g_h/m/l(a : actor, b : actor, g : goal)
can_depend_on_gt_h/m/l(a : actor, b : actor, g : gtype)

Table 7.3: Safety critical systems design: primitive predicates

In Section 7.2.3 the application of our planning-based risk-aware approach will be presented.

7.2.2 Planning domain

Table 7.3 presents the predicates used to describe the *initial state of the system* in terms of actor and goal properties, and social relations among actors:

- `criticality_h/m/l` and `criticality_gt_h/m/l` predicates represents the criticality of the goal, one of **high**, **medium**, or **low**. The criticality level implies the minimum needed level of trust between the actors between which the goal is delegated. For instance, if the criticality of the goal **G** is **high**, then it could be delegated from the actor **A**₁ to the actor **A**₂ only if **A**₁ can depend on **A**₂ for **G** (or for the type of goals which **G** belongs to) with the **high** level of trust.
- `can_depend_on_g_h/m/l` and `can_depend_on_gt_h/m/l` predicates mean that the fulfillment of a specific goal or of any of a specific type can be delegated between the actors, and the trust level of the dependency between these actors for this specific goal or goal type is **high**, **medium**, or **low**, respectively.

All other predicates have the same meaning as in the basic set of predicates introduced in Table ??.

A plan, constructed to fulfill the goals, can contain the following actions, defined in terms of preconditions and effects, expressed with the help of the above predicates.

- **Satisfy**. This action is equivalent to **goal satisfaction** action defined in Section 4.2.
- **AND/OR-Decompose**. This action is equivalent to **goal AND/OR-decomposition** action defined in Section 4.2.
- **Delegate**. The delegation can only take place if the level of trust between the actors is not lower than the criticality level required for the goal to be delegated.
- **Relax**. If there is no way to find a dependency relation which satisfies the required level of trust, then the goal criticality might be relaxed (i.e., lowered). To minimize

Algorithm 1 Planning and evaluation process

```

Require: domain {domain description in PDDL}
           problem {goal and initial state of the problem in PDDL}
           whitelist {a list of allowed action}
           relax{allow/not relaxation}
1: boolean finish ← false
2: while not finish do
3:   plan ← run_planner(domain, problem, relax)
4:   if not evaluate_sat(plan) then
5:     refine_sat(plan, problem)
6:   else if relax and not evaluate_act(plan) then
7:     refine_act(plan, problem, whitelist)
8:   else
9:     finish ← true
10:  end if
11: end while

```

the risk, as soon as the delegation has been performed, the goal criticality is restored to the original value.

The domain *axioms* concerning the propagation of goal satisfaction along goal refinement are defined exactly the same as in Section 4.3.

The presented planning domain was specified in PDDL 2.2 [46] and tested with LPG-td planner [90].

7.2.3 Evaluation process

After a candidate plan is generated by the planner, it should be evaluated based on certain criteria, and then either modified, or approved/declined by the designer. In case of risk-based evaluation metrics, modifying a plan means identifying the actions that should be avoided to lessen risk and then replanning under the new constraints.

In this framework, we consider two types of risk. The first type is a risk of goal satisfaction, called satisfaction risk (**sat-risk**). **Sat-risk** represents the risk of a goal being denied/failed when an actor attempts to fulfill it. The value of the risk can be one of the following: *FD* (Fully Denied), *PD* (Partially Denied), and *ND* (Not Denied). These predicates are adapted from [108], and represent the high, medium, and low level of **sat-risk**, respectively. The second type of risk is related to the risk of goal delegation. It is based on the requirement that the level of trust between two actors should match the criticality of the delegated goal. For instance, if a link between two agents is **highly** trusted, than it can be used for delegating the goals of any criticality level, but if the level of trust of a delegation link is **medium** than only the goals with **low** and **medium** criticality can be delegated along this link, otherwise the risk is introduced.

The process of selecting a suitable design alternative is illustrated in Algorithm 1, which should be run twice. In the first execution, the algorithm constructs a plan without any relaxation actions (i.e., *relax*=false). If there is no solution then the second execution

is run, in which relaxation actions are allowed. Some steps in the algorithm are fully automated (e.g., *run_planner* line 1), while some still need a human involvement (e.g., adding the allowed actions to the *whitelist* in line 1). The algorithm is iterative and comprises the following phases: planning, evaluation, and, finally, plan refinement. There are two evaluation steps in the algorithm: STEP-1 evaluates the risks of goal satisfaction (line 1), and STEP-2 evaluates relaxation actions (line 1). The first execution includes only STEP-1, and if the second execution is necessary, both STEP-1 and STEP-2 are executed. Each evaluation step is followed by a refinement action (line 1 or 1), which aims at changing the planner input so that during the next iteration it will produce the better (i.e. less risky) candidate plan. In the following we give details on the two evaluation steps of the algorithm.

STEP 1: Goal satisfaction evaluation

After a candidate plan is elicited (line 1), it should be evaluated and refined, so that it meets the requirements imposed on it (i.e., the level of risk associated with the plan is below the predefined threshold). The aim of the first evaluation step (line 1 of the Algorithm) is to assure that **sat-risk** values of the candidate plan, i.e. the likelihood of each system goal being denied/failed, do not exceed the thresholds, specified by a designer.

By examining the candidate plan, the goal model of each top goal can be constructed, which shows how a top goal is refined into atomic tangible leaf goals. Here tangible means that for each leaf goal there is an actor that can fulfill it (see e.g. Figure 7.7). Starting from the **sat-risk** values of leaf goals, the risk values are propagated up to the top goals with the help of so called forward reasoning. Forward reasoning is an automatic reasoning technique introduced in [65], which takes a set of **sat-risk** values of leaf goals as an input. Note, that **sat-risk** value for a goal depends on also on which actor satisfies it according to a candidate plan. The values assigned to the leaf goals are propagated along the goal tree up to the top goal, and thus the corresponding value for the top goal is calculated.

If **sat-risk** of one of the top goals is higher than the specified threshold, then the refinement process needs to be performed. The refinement (line 1) identifies those assignments of the leaf goals to actors that should be avoided in order to have the **sat-risk** values of the top goals within the specified thresholds. The refinement process starts by generating a possible set of assignment (i.e., **sat-risk** values of the leaf goals) that results in the top goals having the **sat-risks** below the specified thresholds. This set of assignments is called a reference model. Basically, a reference model is such a set of maximum **sat-risk** values of leaf goals that the resulting **sat-risks** of the top goals do not violate the thresholds. If the **sat-risk** values of leaf goals in the goal model are below the maximum specified in the reference model, then the **sat-risk** of the top goals are acceptable. The reference model can be obtained automatically using backward reasoning [108], which aims at constructing the assignments of leaf goals to actors so that the specified **sat-risk** values for the top goals are achieved. This is done by encoding a goal model as a satisfiability (SAT) problem, and then using a SAT solver to find the possible assignments that satisfy the SAT formula.

By comparing the **sat-risk** values of leaf goals in the goal model with the corresponding values in the reference model, those leaf goals are identified that have higher **sat-risk** values than the corresponding values in the reference model. The problem definition is refined

so that in the next planning iteration the risky satisfaction actions are avoided.

STEP 2: Action evaluation

The second evaluation step (line 1 of the Algorithm) is performed to guarantee that the relaxation actions in a candidate plan are acceptable in terms of risk. We assume that relaxing the criticality of a goal from high to medium, or from medium to low, can be done safely only by the owner of a goal. We say that goal G is owned by A if initially $\text{wants}(A, G)$ holds; in this case all the subgoals of G are also said to be owned by A . To allow a non-owner to relax the criticality of a goal, this goal should be explicitly added to the *whitelist* by a designer.

For instance, in the ATM case study SU1-1 intends to increase his airspace capacity in response to the traffic increase by **delegating his airspace** (G_{11}) to SU1-2. As the fulfillment of G_{11} is critical (the criticality level is **high**), SU1-1 needs to have **high** trust level towards SU1-2 for delegating G_{11} (i.e., $\text{can_depend_on_gt_h}(\text{SU}_{1-1}, \text{SU}_{1-2}, G_{11})$ should hold). Then, SU1-2 refines G_{11} into the subgoals **control the aircraft** (G_2) and **manage the airspace** (G_3). For satisfying these goals, SU1-2 needs to depend on the controller C1-2 for G_2 , and on the planner P1-2 for G_3 . In case the trust level of the dependency of SU1-2 towards C1-2 for G_2 is **medium**, SU1-2 (even though he is not the owner of goal G_2) needs to relax the criticality of G_2 so that it can be delegated to C1-2. This action will *not* be considered critical only if G_2 has been added to the *whitelist* by a designer.

Notice that relaxation actions are introduced only in the second run of algorithm. During the refinement phase (line 1) the problem definition is changed to meet this requirement, and the replanning is performed.

7.2.4 Case study: experimentations

In this section, we illustrate the application of our approach to the ATM case study. The following subsections detail the case study formalization, and the planning-and-evaluation process, performed in accordance with Algorithm 1. The aim of the process is to elicit an appropriate plan for SU1-1's sector, taking into account the constraints and the risks of each alternative. The scenario starts with the intention of SU1-1 to **increase the capacity of airspace** (G_6) as a response to the air traffic increase in sector 1-1. SU1-1 faces a problem that C1-1 is not **available to control** (G_{14}) more traffic. Therefore, SU1-1 needs to modify sector 1-1 without involving C1-1 so that the air traffic increase can be handled.

Case study formalization

The following inputs should be provided for Algorithm 1:

- A formalized problem definition, which contains all the properties of the actors of the ATM system, and their goals. The complete list of properties can be found in Table 7.5.
- Goals of the planning problem (e.g., satisfy G_6 without involving C1-1 in satisfying G_{14}).

- A list of authorized further relaxation actions (*whitelist*).
- Risk values of goal satisfaction actions. Table 7.4 shows all **sat-risk** values of the satisfaction actions.
- Accepted risk values (e.g., risk value of G_6 is at most *PD*).

In Table 7.4 the goal criticality values are presented in column Crit. Goal criticality (H, high, M, medium, or L, low) denotes a minimum level of trust between two actors that is needed if one actor decides to delegate the goal to another actor. For instance, goal **manage airspace** (G_3) is categorized as a *highly critical* goal, and goal **analyze air traffic** (G_8) has *low* criticality. Thus, these goals require different level of trust for being delegated to another actor.

Also, Table 7.4 presents **sat-risk** values of (goal, actor) pairs. **sat-risk** takes one of the tree values: *FD* (Fully Denied), *PD* (Partially Denied), or *ND* (Not Denied). For instance, G_{19} can be satisfied either by actor P1-1, P2-1, or P1-2, and the **sat-risk** value is different for these actors: *full*, *partial*, and *none*, respectively. The empty cells in Table 7.4 means that the actor does not have capabilities to fulfill the corresponding goal.

Goal \ Actor			C1-1	C2-1	P1-1	P2-1	SU1-1	C1-2	P1-2	SU1-2
			Id.	Description	Crit.					
G1	Manage Aircraft within ACC									
G2	Control Aircraft	H								
G3	Manage Airspace	H								
G4	Manage Flight Data	M						PD		
G5	Maintain Air Traffic Flow in Peak-Time									
G6	Increase Airspace Capacity									
G7	Analyze Air Traffic	L								
G8	Re-sectorize within ACC									
G9	Delegate Part of Sector									
G10	Define Schema Delegation	M			ND					PD
G11	Delegate Airspace	H								
G12	Have Controlling Resources									
G13	Have Capability to Control the Aircraft		ND	PD				PD		
G14	Avail to Control		FD	ND						
G15	Have Control Working Position for Controller	H					ND			PD
G16	Have Authorization for FD Modification	M	ND	ND						
G17	Have Capability to Manage FD				ND	PD				
G18	Have Resources for Planning	M					ND			
G19	Have Capability to Manage Airspace				FD	PD			PD	
G20	Have Capability to Analyze Air Traffic				PD					
G21	Avail to Plan				ND	ND			ND	
G22	Have Control Working Position for Planner	H					ND			ND

Table 7.4: ATM case study: goal criticality and **sat-risk**

Table 7.5 presents actor capabilities (**can_satisfy**), possible ways of goal refinements (**decompose**), and possible dependencies among actors (**can_depend_on**) together with the level of trust for each dependency. For instance, actor SU1-1 **can_satisfy** goals G_{15} , G_{18} , and G_{22} , and the actor has knowledge to decompose G_1 , G_5 , G_6 , G_8 , and G_9 . SU1-1 has *high* level of trust towards C1-1 and C2-2 for G_2 .

Actor	can_satisfy	decompose			can_depend_on		
		type	top-goal	sub-goals	level	dependum	dependee
SU1-1	G15	And	G1	G2, G3	H	G2	C1-1, C2-1
	G18	And	G5	G6, G7	H	G3	P1-1, P2-1
	G22	Or	G6	G8, G9	H	G4	P1-1, P2-1
		And	G8	G2, G3, G4	M	G7	P1-1
		And	G9	G10, G11	M	G10	P1-1
					L	G10	SU1-2
				H	G11	SU1-2	
P1-1, P2-1	G17	And	G3	G18, G19	H	G22	SU1-1
	G19	And	G4	G16, G17, G18			
	G21	And	G7	G18, G20			
P1-1	G10				L	G16	C1-1
	G20						
P1-2				L	G16	C2-1	
C1-1, C2-1	G13	And	G2	G4, G12, G13	H	G15	SU1-1
	G14	And	G12	G14, G15			
	G16						
C1-1				M	G4	P1-1	
C2-1					M	G4	P2-1
					M	G4	P1-1
SU1-2	G10	And	G11	G2, G3	M	G2	C1-2
	G15				M	G3	P1-2
	G22						
P1-2	G19	And	G3	G19, G21, G22	M	G22	SU1-2
	G21						
C1-2	G4	And	G2	G4, G13, G15	M	G15	SU1-2
	G13						

Table 7.5: ATM case study: actor and goal properties

Planning and evaluation process

STEP 0: Planning. After specifying the inputs, the planner is executed to elicit a candidate plan to fulfill the predefined goals, which is shown in Figure 7.6(a). These goals state that the plan should satisfy G_6 , and the solution should not involve C1-1 to satisfy G_{14} because C1-1 is already overloaded controlling the current traffic. Moreover, the planner should avoid involving the other ACC (i.e., SU1-2) by avoiding the delegation of G_{11} to SU1-2 even it is possible in Table 7.5. Before adopting the candidate plan (Figure 7.6(b)), two evaluation steps explained in previous section should be performed to ensure the risk of the candidate plan is acceptable.

STEP 1: Goal satisfaction evaluation assesses the satisfaction risk of a candidate plan. The goal model of goal G_6 (in Figure 7.7) is constructed on the basis of the candidate plan (in Figure 7.6(b)). The goal model shows which actors are responsible for satisfying the leaf goals. For instance, G_{19} must be satisfied by P1-1 and, moreover, in this scenario, G_9 is left unsatisfied because the other or-subgoal, G_8 , was selected to satisfy G_6 .

In this scenario, we assume that the acceptable **sat-risk** value for G_6 is PD . To calculate the **sat-risk** value of goal G_6 , forward reasoning is performed (i.e., the **sat-risk** values of leaf goals in Table 7.4 are propagated up to the top goal). This reasoning mechanism is a part of the functionality of the GR-Tool⁶, a supporting tool for goal

⁶<http://sesa.dit.unitn.it/goaleditor>

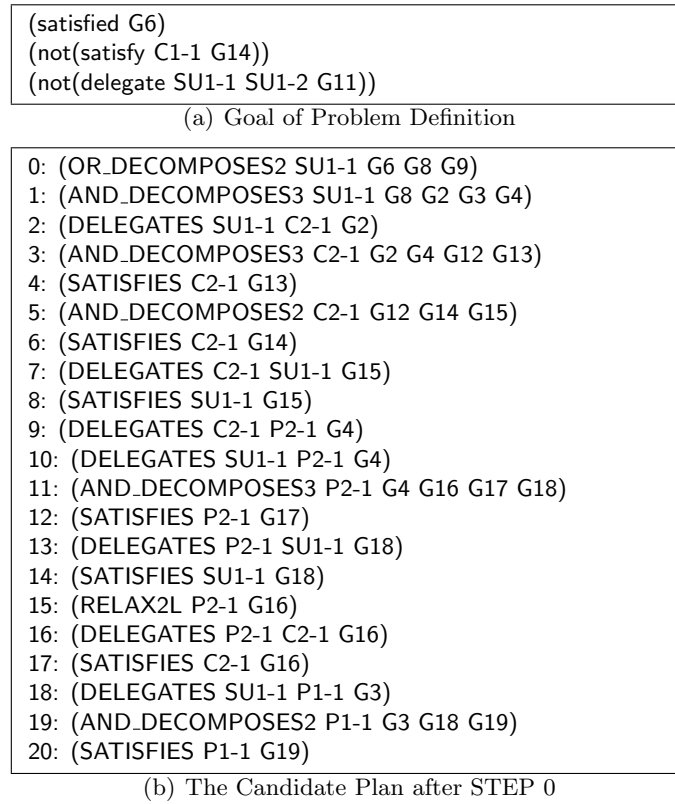


Figure 7.6: ATM case study: plan for increasing air space capacity

analysis. By means of the forward reasoning, we obtain that the *sat-risk* of G_6 is FD , which is higher than the acceptable risk (i.e., PD). Thus, the refinement is needed to adjust the problem definition, so that a less risky plan is constructed during the next replanning. The refinement starts with the elicitation of a reference model using backward reasoning. The reference model specifies that all leaf goals must have at most PD *sat-risk* value in order the *sat-risk* of top goal G_6 not to be higher than PD .

By comparing the *sat-risks* of leaf goals in the goal model with the reference model, G_{19} (satisfied by P1-1) is detected to be a risky goal; its *sat-risk* (in Table 7.4) is FD which is higher than the one in the reference model. Therefore, the problem definition is refined to avoid P1-1 satisfying G_{19} . As G_{19} is a subgoal of G_3 , the decomposition action is also negated, as the previous related action, according to the procedure explained in the previous section. Thus, the problem definition is refined, and the goal of the planning problem is now of the form shown in Figure 7.8(a). Afterwards, the planner is run to elicit a new candidate plan. Basically, the new candidate plan is almost the same with the previous plan (Figure 7.6(b)), the only difference is in lines 18-20 (see Figure 7.8(b)). Later, this candidate plan is evaluated by going through the next step to ensure all the actions (especially, further relaxations) are acceptable in terms of risks.

STEP 2: Action evaluation filters the malicious relaxation actions. The scenario starts from the goal G_6 which is wanted by SU1-1. As all the other goals of the candidate

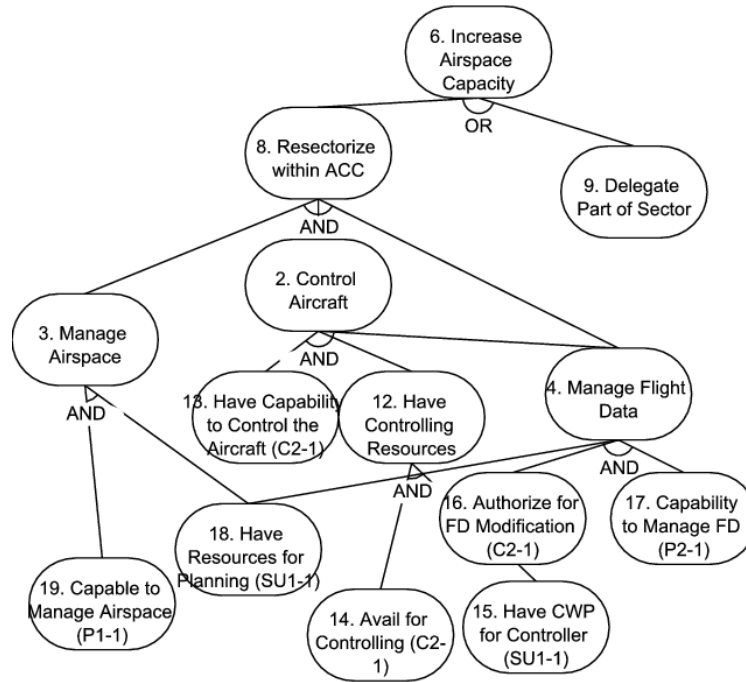


Figure 7.7: ATM case study: goal model for a candidate plan of Figure 7.6(b)

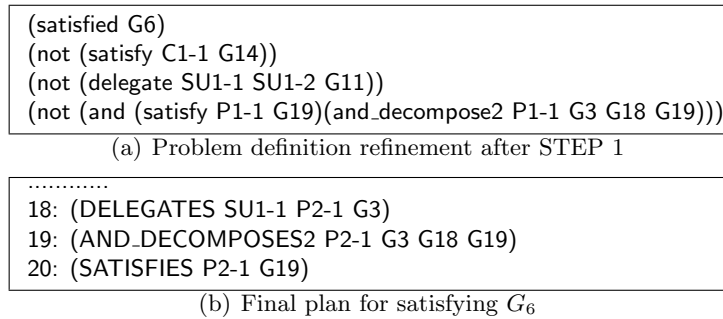


Figure 7.8: ATM case study: final problem definition and plan

plan are the result of the refinement of G_6 , the owner of all of them is again SU1-1. Thus, relaxing the criticality of any goal that is performed by any actor except SU1-1 is seen as a risky action.

For instance, P2-1 relaxes the criticality of G_{16} (line 15 in Figure 7.6(b)) to **low** instead of **medium**. By default this action is a risky one and should be avoided, unless the designer states explicitly that this action is not risky by adding it to the *whitelist*. Once it is considered unacceptable, the goal of the planning problem should be extended with the negation of the relaxation action (i.e., $(\text{not } (\text{relax2l } P2-1 G1))$).

Moreover, the designer can also introduce rules to avoid certain actions. For instance, the designer may prevent C2-1 from delegating G_4 to P2-1 (line 15 in Figure 7.6(b)) by adding a new predicate to the goal of the planning problem (namely,

(not (delegate C2-1 P2-1 G4))). For the sake of simplicity all the possible relaxation actions in the candidate plan are put to the *whitelist*, so we do not refine the problem definition any further.

Thus, the last candidate plan to redesign SU1-1's sector is approved s.t. the traffic increase can be handled. Moreover, the plan is guaranteed to have risk values less/equal than the predefined thresholds (i.e., *sat-risk* of G_6 is less or equal than PD).

.....

7.3 Runtime application: self-configuring systems

One of the characteristic features of socio-technical systems (STS), challenging from a design point of view, is that their structure has to evolve dynamically in response to the changes of the environment. When new requirements are introduced, when an actor leaves the system or when a new actor comes, the system structure needs to be redesigned and revised. In this section, we present an approach to dynamic reconfiguration of a socio-technical system structure in response to internal or external changes. The approach is based on the ideas presented in Chapters 4 and 5, namely, it uses planning for generating possible alternative configurations, and local strategies for their evaluation. In the following, we present a reconfiguration mechanism, which aims at making a socio-technical system self-configuring, and apply it to a simple case study.⁷

An important aspect of a socio-technical system is its dynamicity: an STS operates in continuously changing environments and, accordingly, its structure changes dynamically. This calls for the new type of requirements that introduce the need of highly adaptable and reconfigurable systems [117].

Recently, a lot of work has been devoted to the problem of dynamic reconfiguration and adaptation of software systems [45, 34, 15, 75, 43, 118]. Attempts to adjust the existing agent-oriented methodologies, such as Gaia [34], or to create a specialized ones, such as Adelfe [15], to develop adaptive agents are described in the literature. All these proposals can be grouped in approaches that consider the reconfiguration process from the local and from the global perspective.

Self-configuration from the local perspective, i.e. on the level of an individual agent, is related to the concept of self-organization. Self-organization phenomena (see e.g. [124]) is observed when some macroscopic system properties arise (emerge) dynamically from the local micro-level interactions. However, such perspective is sometimes not enough as it does not allow to reach all the desired properties of an STS which works in the dynamic environment [45]. For example, the social behavior of being helpful, or following the imposed external laws, is difficult to describe by the "individual rationality" principle assumed by self-organization emergent models. Another example is a scientific institution, which could hardly function on the base of self-organization principles, without any centralized authority. Differently, we strongly agree on the approach presented in [45], which suggests combining the perspective of individual agents with the global one, in which reconfiguration is controlled centrally.

⁷The material presented in this section was published in [26].

In this section, we propose an approach to the problem of dynamic reconfiguration of an STS structure in response to the internal and/or environmental changes. The approach is based on planning techniques for exploring the space of alternative configurations, combined with evaluating the generated alternative in terms of local strategies of the system actors. The approach comprises the following steps.

- Identify system actors, their goals, capabilities, and interrelations.
- Select the initial configuration by the following three-step iterative procedure:
 - construct the assignment of goals to actors with the help of planning, so that all goals are to be satisfied;
 - evaluate the obtained assignment with respect to local interests of the system actors, in order to identify which actors will be motivated to deviate from the assignment;
 - in case the deviation is inevitable, reformulate the planning problem, and go to the construction of the next assignment.
- Monitor the STS and the environment, in case of changes assess whether the reconfiguration is necessary. Reconfigure the system with the help of the above described iterative procedure.

There exists a number of works which are, to some extent, similar to ours. [45] deals with the problem of dynamic reorganization of agent societies, and presents the classification of reorganization situations. According to the authors, the paper is exploratory in nature, and contains the discussion of the problem rather than the possible solutions. In [75] *Moise+* organizational model is extended to support the reorganization of multi-agent systems. The organization is represented along its structural and functional dimensions, and the deontic relation among these dimensions is defined. The reorganization process is performed by the set of organization agents playing the specific roles, such as *OrgManager* that is in charge of managing the reorganization process, *Monitor* that is monitoring the system, *Designer* that develops reorganization proposals, and the like. However, no specific guidelines are provided on how the new system configuration should be conducted. [43] describes how to design adaptive multi-agent systems using the organizational model that consists of a structural and state models of an organization, and a transition function from one organizational state to another. The structural model contains the information about goals, agent roles, organizational rules and laws. A state model is an instance of an organization which includes a set of agents together with the relationships between them and other structural model components. A number of events called reorganization triggers are described, which may cause the system reorganization. The reorganization process is assumed to be application specific, and the selection of an appropriate configuration relies on maximizing a sort of utility function, so called organization capability score. In [118] the techniques for organization and reorganization of multi-agent systems in the domain of oceanography are presented. The reorganization is domain specific, and is based on communication protocols, with the help of which groups of agents cooperate and reorganize themselves in response to the environmental changes.

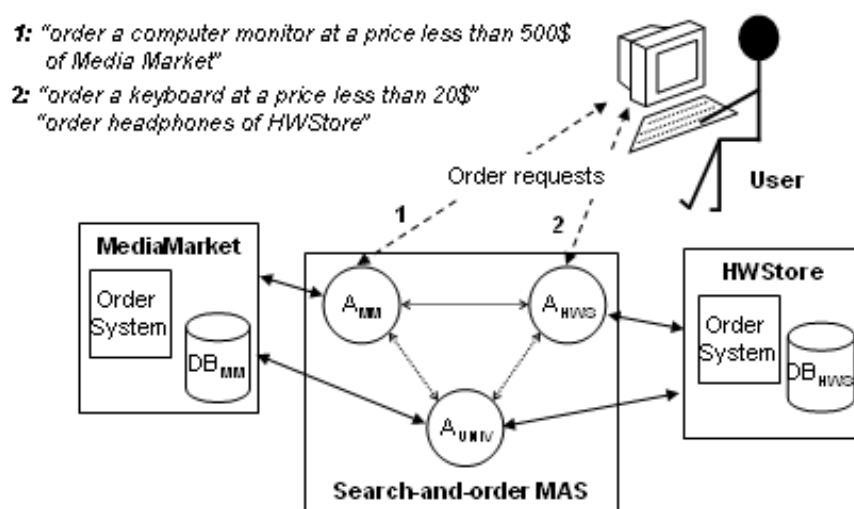


Figure 7.9: Search-and-order MAS

Our differs from the above described work as it provides, independently of the domain, the concrete guidelines on how the reorganization process could be organized. Another important point is the automation support: the process of exploring the space of alternative system configurations is performed automatically with the help of planning techniques. Also the local strategies of an STS actors are taken into account, which allows each actor to evaluate the new configuration from the local perspective, and deviate from it if the load/risk/complexity of the new assignment is unacceptable for this actor. Taking the local strategies of the system actors into consideration makes our approach particularly useful for the socio-technical systems, because strategic and rational behavior is intrinsic to the human actors.

In the following, we first present a motivating example, then introduce the reconfiguration mechanism, illustrate it with the example, and discuss the architectural and implementation issues.

7.3.1 A motivating case study

The case study we present in this section was selected to be quite simple for the reasons of compactness and ease of understanding.

Consider a small firm which sells office equipment to its customers. The office equipment is supplied by two companies, *MediaMarket* and *HWStore*, both having a database containing information about supplied goods, their technical characteristics and prices. To organize the placing of orders for the sell items, the supporting software system of the firm comprises a subcomponent, called *search-and-order multi-agent system (MAS)*, which consists of three agents, see Figure 7.9. These agents can process the user orders, i.e. search for the required item in the supplier's database, provide information to the customer if the item was found, and formulate the request to a supplier otherwise.

Two of these agents, A_{MM} and A_{HWS} , can work only with the database of one supplier, *MediaMarket* and *HWStore*, respectively. In other words, because of the (limited) capabilities A_{MM} possess, it cannot work with *HWStore* database, and vice versa. The third agent A_{UNIV} is a reserve one, it can query both databases, although, less efficiently than A_{MM} and A_{HWS} , and is used when other agents are unable to hold numerous requests of a user (a clerk of a firm). In principle, A_{UNIV} can be a human agent which is exploited only if some critical situation occurs: satisfying the customer's request needs some specific human support (e.g. making a phone call), or the software component fails and the customer remains unserved, which violates the organizational rules, etc. However, as our approach treats both artificial and human actors in a similar way, we will not introduce human actors in the example due to the space and simplicity reasons. Also note that the number of suppliers is limited to two just for the sake of simplicity. In reality in such a system there can be tens, or even hundreds of different suppliers, and a limited number of agents, each with different set of capabilities. Some of these agents are more efficient when working with one specific "type" of suppliers, while others are "universal", i.e. can work with all suppliers. The task of allocating the incoming orders in a (sub)optimal way is indeed challenging, and this is what we are going to automate with the help of the approach proposed in this section.

Suppose that initially, i.e. at time point t_0 , there are three requests the agents have to satisfy. One query, *order a computer monitor at a price less than 500\$ of MediaMarket*, is sent by a user to A_{MM} , and two, *order a keyboard at a price less than 20\$* and *order headphones of HWStore*, sent to A_{HWS} . Even for this simple example there are a number of alternative initial configurations. E.g., the query *order a keyboard at a price less than 20\$* could be accomplished by searching either in the database of *MediaMarket* or *HWStore*. Another source of alternatives is whether to involve A_{UNIV} in performing the queries or not. Thus, the problem is how to assign queries to agents in a (sub)optimal way.

7.3.2 Planning domain

To define our domain, we use a subset of the basic set of predicates introduced in Table ???. Namely, predicates take variables of three types: actors, goals and goal types. To typify goals, `type(g : goal, gt : gtype)` predicate is used. Actor capabilities with respect to goal types are described with `can_satisfy_gt(a : actor, gt : gtype)` predicate. Social dependencies among actors are reflected by `can_depend_on(a : actor, b : actor)` predicate, and predefined ways of goal refinement – by `and/or_decompositionn(g : goal, g1 : goal, . . . , gn : goal)` predicates. The initial desires of actors are represented with `wants(a : actor, g : goal)` predicate. When a goal is fulfilled, `satisfied(g : goal)` predicated becomes true for it.

As in Section ??, a plan, constructed to fulfill the goals of the system actors, comprises *goal satisfaction*, *goal delegation* and *goal decomposition* actions. LPG-td [90] planner with PDDL [46] as a specification language are used to implement the planning domain.

The formalization of the initial system configuration is presented in Figure 7.10. The plan P_0 generated by the planner is presented in Figure 7.11, and its graphical representation is depicted in Figure 7.12. Note, that there are several alternative configurations in which all goals are satisfied (e.g. the goal *OrderKeyboard* can be achieved by A_{MM} by

```

type (OrderMonitorOfMM, tDB1)
type (OrderKeyboardOfMM, tDB1)
type (OrderKeyboardOfHWS, tDB2)
type (OrderHeadphonesOfHWS, tDB2)
can_depend (AMM, AHWS)   can_depend (AHWS, AMM)
can_depend (AMM, AUNIV)   can_depend (AUNIV, AMM)
can_depend (AHWS, AUNIV)   can_depend (AUNIV, AHWS)
can_satisfy_gt (AMM, tDB1)
can_satisfy_gt (AHWS, tDB2)
can_satisfy_gt (AUNIV, tDB1)
can_satisfy_gt (AUNIV, tDB2)
or_decomposition2 (OrderKeyboard, OrderKeyboardOfMM, OrderKeyboardOfHWS)
wants (AMM, OrderMonitorOfMM)
wants (AHWS, OrderHeadphonesOfHWS)
wants (AHWS, OrderKeyboard)

```

Figure 7.10: Search-and-order MAS: planning problem formalization

```

(SATISFIES AMM OrderMonitorOfMM)
(SATISFIES AHWS OrderHeadphonesOfHWS)
(OR_DECOMPOSES AHWS, OrderKeyboard
  OrderKeyboardOfMM OrderKeyboardOfHWS)
(SATISFIES AHWS OrderKeyboardOfHWS)

```

Figure 7.11: Search-and-order MAS: initial configuration

satisfying the *OrderKeyboardOfMM* or-subgoal, instead of *OrderKeyboardOfHWS*; or the goal *OrderHeadphonesOfHWS* can be delegated to A_{UNIV} and satisfied by it).

The configuration presented in Figure 7.11 is just the *first* one generated by the planner, that is, it is a starting point to be evaluated and, possible, amended. We use here the cost-based planning-and-evaluation procedure presented in Section ??, with the overall goal of balancing the work load imposed on the search-and-order MAS actors.

We assume that different order queries have different costs for the three system agents, depending, e.g., on the complexity of a query. The costs for A_{UNIV} are higher, which is caused by its “universality”, i.e. ability to work with both suppliers. Moreover, the order queries are subdivided into two classes: “simple” and “complicated”. The cost of

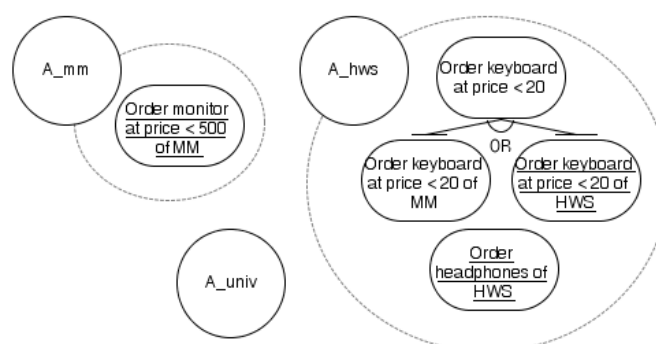


Figure 7.12: Search-and-order MAS: initial configuration

the satisfaction of a simple query is lower than a corresponding cost for the complicated query. An example of a complicated query could be *order best-selling HDD of HWStore*, as it requires obtaining the statistical information.

The cost of any delegation is equal to 1 unit, the cost of a decomposition is equal to 2. For A_{MM} , A_{HWS} performing simple order queries costs 10 units, performing complicated ones – 15, for A_{UNIV} – 15 and 20, respectively. Tolerable bound of load for all three agents, under which they are not willing to deviate from the imposed reconfiguration plan, is equal to 30 units. In these conditions the costs of the obtained initial configuration plan P_0 (see Section 2.2) are the following: $c(A_{MM}, P_0) = 10$, $c(A_{HWS}, P_0) = 2 + 10 + 10 = 22$, and $c(A_{UNIV}, P_0) = 0$. Due to simplicity of the example, this first solution generated by the planner is satisfactory from the point of view of all three agents, i.e. the evaluation shows that the plan costs are within the tolerable load bounds for each agent.

7.3.3 Reconfiguration mechanism

In this section, we present a centralized reconfiguration mechanism, based on the planning-and-evaluation procedure described in Section ??.

The reconfiguration mechanism

- collects and manages the information about the system;
- evaluates both the overall system load, and the local utilities of each actor to decide whether the system needs to be redesigned in response to external or internal changes;
- and, if the above evaluation shows that the reconfiguration is needed, replans the system structure in order to optimize the distribution of load imposed on system actors.

It stores and updates

- the current problem definition *problemDef*, i.e. *actor* and *goal* variables, and predicates describing them;
- the list of all goals $G = \{g_i, i = 1..n\}$ present in the system together with their states (described in the next paragraph), also for each goal the actor who initially wanted it to be satisfied is stored;
- the current plan of actions, i.e. a list of actions $D = \{d_j, j = 1..m\}$ generated during the last (re)design iteration and not accomplished so far;
- archived data, e.g. *actionLog*.

To describe the states of the goals in G , we use the two of already introduced predicates, namely, *wants*($a : actor, g : goal$) and *satisfied*($g : goal$). In addition, we introduce a predicate *committed*($a : actor, g : goal$). Predicate *committed*(a, g) becomes true when a reconfiguration mechanism is notified that a has committed to g , meaning that a has taken a decision to satisfy g . This predicate is used to support the *minimal change* principle

```

0  if notified(notifMess) and  $t_{curr} - t_{prev} \leq \phi$  quit
1  if notifMess = "a has committed to g" then
2       $G : committed(a, g) \leftarrow true$ 
3      move Decompose(..., g, ...) and Delegate(..., g, ..) from D to ActionLog
4  if notifMess = "a has achieved g" then
5       $G : committed(a, g) \leftarrow false, satisfied(g) \leftarrow true$ 
6      move Satisfy(a, g) from D to ActionLog
7      if  $load(a) \leq freeEnoughBound$  then
8          ReplanWithG( $\emptyset$ ) /* with empty set of new goals */
9  if notifMess = "g is removed" then
10     if g is a subgoal of a valid goal then quit /* abnormal situation */
11     remove variable g and predicates containing g from problemDef
12     move actions containing g from D to removedActionLog
13     for each subgoal  $g_{sub}$  of g, s.t.  $g_{sub}$  is not a (subgoal of any) valid goal
14         repeat lines 11–12
15     for each actor a if  $load(a) \leq freeEnoughBound$  then
16         ReplanWithG( $\emptyset$ ) /* with empty set of new goals */
17     exit for each loop
18 if notifMess = "a leaves the system" then
19     for each g initially wanted by a /* information stored in G */
20         repeat lines 11–12
21     remove variable a and predicates containing a from problemDef
22     move actions containing a from D to removedActionLog
23     remove predicates containing a from G, put affected goals to newGoalsList
24     Replan(newGoalsList)
25 if notifMess = "new a introduced" then
26     add a variable and related predicates to problemDef and G
27     ReplanWithG( $\emptyset$ ) /* with empty set of new goals */
28 if notifMess = "new g introduced" then
29     add g variable and related predicates to problemDef and G
30     Replan( $\{g\}$ )

```

Figure 7.13: Reconfiguration algorithm

during the reconfiguration process. As it will be seen from the algorithm presented in this section, the reconfiguration does apply to the *committed* goals, and thus, not all the STS structure is revised each time. Note that $satisfied(g)$ implies $\neg committed(a, g)$.

The reconfiguration algorithm is presented in Figure 7.13, and is organized in a way that a block corresponds to one internal or environmental change. The notification about the change is obtained either from the inside of the system or from the environment. Each system actor is obliged to communicate to some central point if it committed to, or achieved a goal. In order to avoid continuous replanning, a time slot ϕ is introduced, such that the triggering events initiate evaluation and replanning only if the time passed since the last replanning is greater than ϕ (**line 0**).

In the following we explain each block briefly.

- (**lines 1–3**) An actor *a* has committed to do a goal *g*. In this case $committed(a, g)$ is set to be true, and all decompositions and delegations of *g* are moved to the action log.

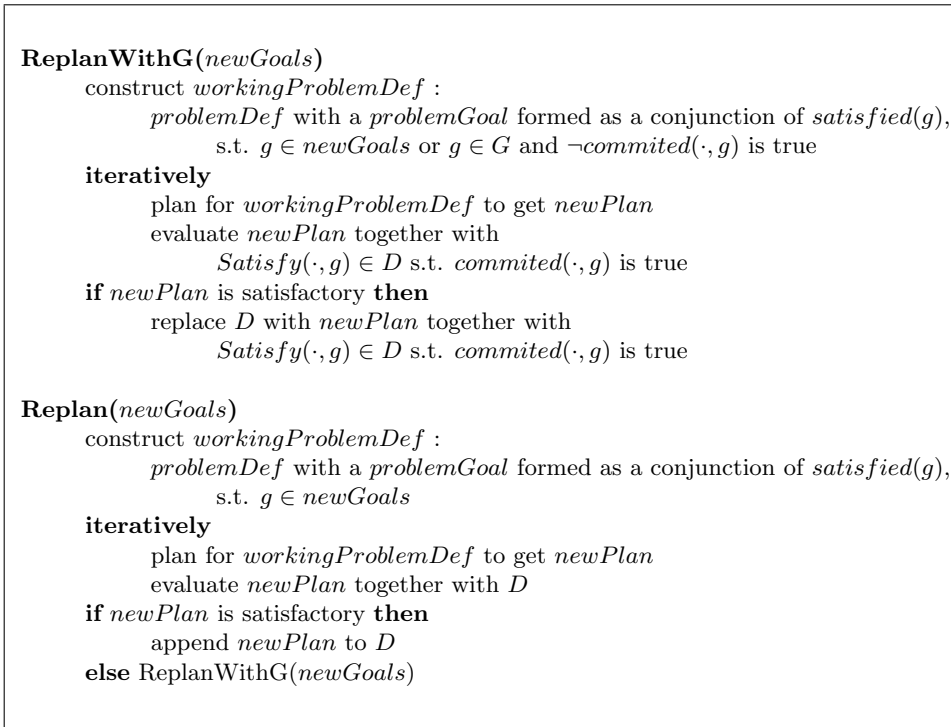


Figure 7.14: Replanning procedures

- (lines 4–8) An actor *a* has achieved a goal *g*. In this case *satisfied*(*g*) is set to be true, and satisfaction action is moved from *D* to the action log. Then it is evaluated whether the actor that has satisfied the goal is “free enough”, in a sense whether the total cost of the actions in *D* it is involved in is less than a predefined threshold. If it is the case, the replanning with non-committed goals procedure *ReplanWithG*, presented in Figure 7.14 and described below, is performed.
- (lines 9–17) One of the imposed requirements is relaxed, i.e. a goal *g* is no longer needed to be achieved. We assume that *g* is not a subgoal of any other goal present in the system. In this case the corresponding variable *g* : *goal* and predicates in which *g* appears are removed from the problem definition. All action containing *g* are moved from *D* to the removed action log. The same “removal procedure” is done for each subgoal of *g*. Then it is evaluated whether any of the system actors is “free enough” in the above defined sense, and if such actors exist, the *ReplanWithG* replanning procedure is performed.
- (lines 18–24) One of the existing actors leaves the system. For each goal that was initially wanted by this actor, the above described “removal procedure” is performed. Then the corresponding variable *a* : *actor* and predicates in which *a* appears are removed from the problem definition, all actions containing *a* are moved from *D* to the action log. All goals which was wanted by *a*, or which *a* was committed to, are considered to be new to the system, and the general replanning procedure *Replan*,

(SATISFIES A_{MM} OrderMonitorOfMM) (SATISFIES A_{HWS} OrderHeadphonesOfHWS) (OR.DECOMPOSES A_{HWS} , OrderKeyboard OrderKeyboardOfMM OrderKeyboardOfHWS) (SATISFIES A_{HWS} OrderKeyboardOfHWS) (SATISFIES A_{HWS} OrderSpeakersOfHWS)
--

Figure 7.15: Search-and-order MAS: first plan at t_1

presented in Figure 7.14 and described below, is performed.

- **(lines 25–27)** A new actor joins the system. In this case a new variable $a : actor$ appears in the problem definition together with the predicates describing the properties of a . Then the *ReplanWithG* replanning procedure is performed.
- **(lines 28–30)** A new requirement to the system has been introduced, i.e. a new goal g is to be satisfied. In this case a new variable $g : goal$ appears in the problem definition together with the predicates describing the properties of g , and for some actor a of the system $wants(a, g)$ becomes true. Then the *Replan* replanning procedure is performed.

In Figure 7.14 the replanning procedures used through the algorithm are introduced.

1. **ReplanWithG.** First planning for all new goals and goals in G , except for committed ones is performed; then the evaluation is done for the intermediate plan – the new plan in which additional actions are included, $Satisfies(a, g) \in D$, such that $committed(a, g)$ is true. If the evaluation is successful, D is replaced with the intermediate plan.
2. **Replan.** Planning is performed for new goals, and then the intermediate plan – D plus the new plan – is evaluated. If the evaluation is successful, new actions are added to D . If not the **ReplanWithG** is performed.

If it is still impossible to find a plan of actions to satisfy the system goals, then the commitments of actors to goals might be revised. However, this feature is not yet supported by our framework, and is not addressed here.

7.3.4 Case study: experimentations

Let us now illustrate the proposed procedure with the help of the Search-and-order MAS example. We consider the reaction of the reconfiguration mechanism to the two triggering events, occurred at the time steps t_1 and t_k .

Step t_1 . Suppose that a new request arrived to the agent A_{HWS} , *order speakers at price between 10 and 30\$ of HWS*, which is simple. Till that moment A_{HWS} has committed to *order headphones of HWStore*.

Replanning only for the new goal gives the result presented in Figure 7.15.

The costs for the obtained plan P_1 are the following: $c(A_{MM}, P_1) = 10$, $c(A_{HWS}, P_1) = 2 + 10 + 10 = 32 > 30$, and $c(A_{UNIV}, P_1) = 0$. As far as A_{HWS} is not satisfied with the imposed load, replanning for all the goals, except committed, is performed.

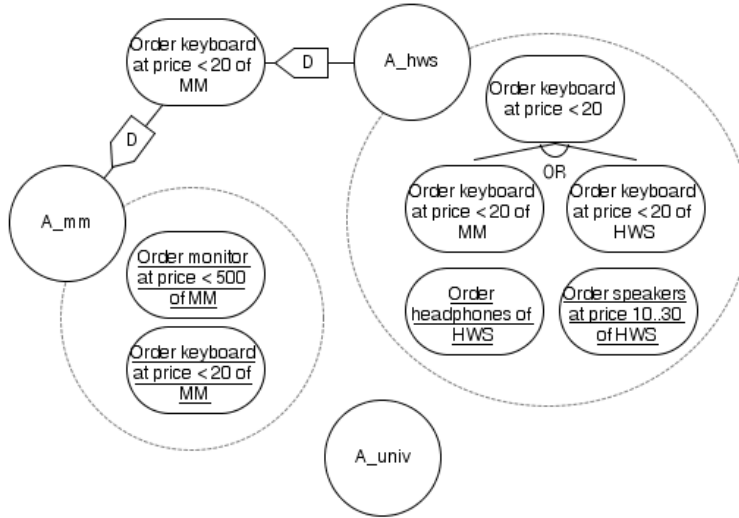
Figure 7.16: Search-and-order MAS: second plan at t_1 

Figure 7.17: Search-and-order MAS: first reconfiguration

The resulting plan P_2 is illustrated in Figure 7.16.

The costs for the P_2 are the following: $c(A_{MM}, P_2) = 10 + 10 = 20$, $c(A_{HWS}, P_2) = 2 + 1 + 10 + 10 = 23$, and $c(A_{UNIV}, P_2) = 0$. As far as all $c(\cdot, P_2) < 30$, the reconfiguration plan P_2 is adopted. The assignment structure is revised, and redesigned as depicted in Figure 7.17.

Step t_k . Suppose that a new request arrived to the agent A_{HWS} , *order best-selling HDD of HWS*, which is classified as complicated. Till this moment A_{MM} has committed to *order a keyboard at a price less than 20\$ of MM*, A_{HWS} has committed to *order headphones of HWS*, and A_{UNIV} has committed to a new simple goal from A_{MM} .

Replanning only for the new goal gives the result presented in Figure 7.18.

The costs for the P_k are the following: $c(A_{MM}, P_k) = 21$, $c(A_{HWS}, P_k) = 23 + 15 = 38 > 30$, and $c(A_{UNIV}, P_k) = 15$. As far as A_{HWS} is not satisfied with the imposed load, replanning for all the goals, except committed is performed.

The resulting plan P_{k+1} is presented in Figure 7.19.

The costs for the P_{k+1} are the following: $c(A_{MM}, P_{k+1}) = 21$, $c(A_{HWS}, P_{k+1}) = 24$, and $c(A_{UNIV}, P_{k+1}) = 15 + 20 = 35 > 30$. As far as A_{UNIV} is not satisfied with P_{k+1} , the replanning is performed.

The resulting plan P_{k+2} is illustrated in Figure 7.20.

```

(SATISFIES  $A_{MM}$  OrderMonitorOfMM)
(OR_DECOMPOSES  $A_{HWS}$ , OrderKeyboard
  OrderKeyboardOfMM OrderKeyboardOfHWS)
(PASSES  $A_{HWS}$   $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{HWS}$  OrderHeadphonesOfHWS)
(SATISFIES  $A_{HWS}$  OrderHDDOfHWS)
(SATISFIES  $A_{HWS}$  OrderSpeakersOfHWS)
(SATISFIES  $A_{UNIV}$  GoalFrom $A_{MM}$ )

```

Figure 7.18: Search-and-order MAS: first plan at t_k

```

(SATISFIES  $A_{MM}$  OrderMonitorOfMM)
(OR_DECOMPOSES  $A_{HWS}$ , OrderKeyboard
  OrderKeyboardOfMM OrderKeyboardOfHWS)
(PASSES  $A_{HWS}$   $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{HWS}$  OrderHeadphonesOfHWS)
(SATISFIES  $A_{HWS}$  OrderSpeakersOfHWS)
(SATISFIES  $A_{UNIV}$  GoalFrom $A_{MM}$ )
(PASSES  $A_{HWS}$   $A_{UNIV}$  OrderHDDOfHWS)
(SATISFIES  $A_{UNIV}$  OrderHDDOfHWS)

```

Figure 7.19: Search-and-order MAS: second plan at t_k

The costs for the P_{k+2} are the following: $c(A_{MM}, P_{k+2}) = 21$, $c(A_{HWS}, P_{k+2}) = 10 + 2 + 1 + 15 + 1 = 29$, $c(A_{UNIV}, P_{k+2}) = 15 + 15 = 30$. As far as all $c(., P_{k+2}) \leq 30$, the reconfiguration plan P_{k+2} is adopted. The assignment structure is revised, and redesigned as depicted in Figure 7.21.

7.3.5 General architecture for self-configuring systems

To implement the presented approach, i.e. to add to a socio-technical system the ability to self-configure, we propose the two-layered multi-agent architecture, presented in Figure 7.22.

The lower layer, which we call the operational layer (OL), is domain-specific, and comprises a set of agents aiming to satisfy the goal of an STS (place the orders to the suppliers, book the plane tickets, manage meeting agenda, etc.). On the upper layer,

```

(SATISFIES  $A_{MM}$  OrderMonitorOfMM)
(OR_DECOMPOSES  $A_{HWS}$ , OrderKeyboard
  OrderKeyboardOfMM OrderKeyboardOfHWS)
(PASSES  $A_{HWS}$   $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{MM}$  OrderKeyboardOfMM)
(SATISFIES  $A_{HWS}$  OrderHeadphonesOfHWS)
(SATISFIES  $A_{HWS}$  OrderHDDOfHWS)
(SATISFIES  $A_{UNIV}$  GoalFrom $A_{MM}$ )
(PASSES  $A_{HWS}$   $A_{UNIV}$  OrderSpeakersOfHWS)
(SATISFIES  $A_{UNIV}$  OrderSpeakersOfHWS)

```

Figure 7.20: Search-and-order MAS: third plan at t_k

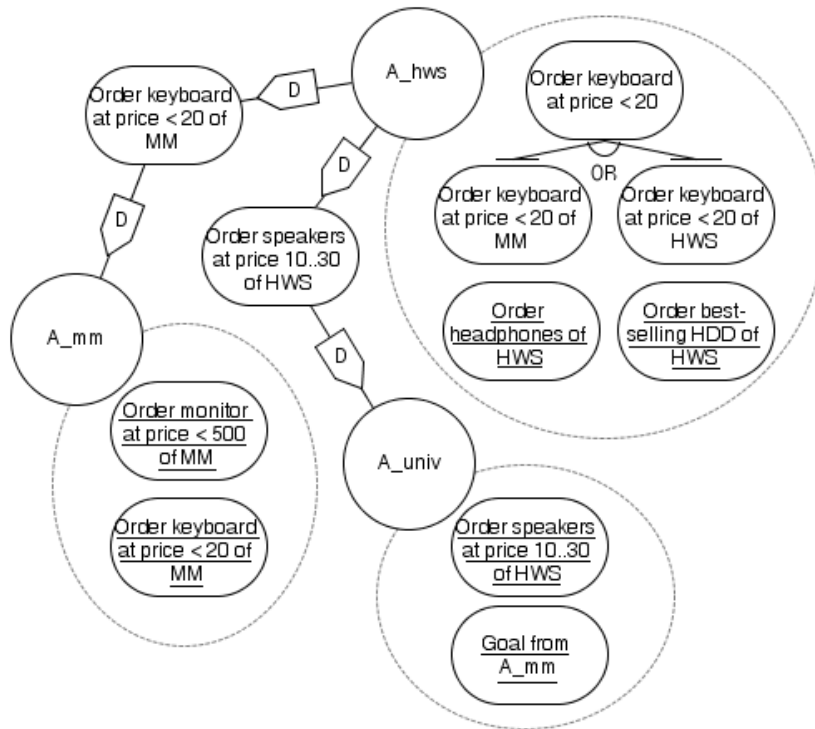


Figure 7.21: Search-and-order MAS: second reconfiguration

which we call the control layer (CL), there sit four agents – *Monitor*, *Controller*, *Planner* and *Evaluator*. *Monitor* is responsible for the communication with the agents of the operational layer, and the environment. The OL agents notify the *Monitor* about the relevant changes. *Controller*, *Planner* and *Evaluator* realize the domain-independent procedures: reconfiguration, planning and evaluation, respectively. The data they store and process (status of system goals, formal definition of a planning problem, costs of actions for each actor) is specific to the given STS. *Controller* is following the reconfiguration mechanism presented above, delegating planning and evaluation tasks to *Planner* and *Evaluator*, respectively. The new system configuration, produced by *Controller*, *Planner* and *Evaluator*, is propagated to the OL agents by *Monitor*. The separation of duties between the control layer agents is detailed in Table 7.6.

We propose that the described multi-agent architecture is to be implemented in JADE (Java Agent DEvelopment framework) [2], FIPA-compliant [3] framework for multi-agent systems development. Four agents of the control layer will have the same functionality for any domain-specific instance of the architecture. The *Controller* agent will implement the reconfiguration algorithm presented in Figure 7.13. The *Monitor* needs to implement the communication with OL agents and the environment (e.g. using standard FIPA protocols, like ContractNetProtocol). The functionality of the *Planner* and *Evaluator* agents has been already implemented in S&D Tropos Tool [109].

.....

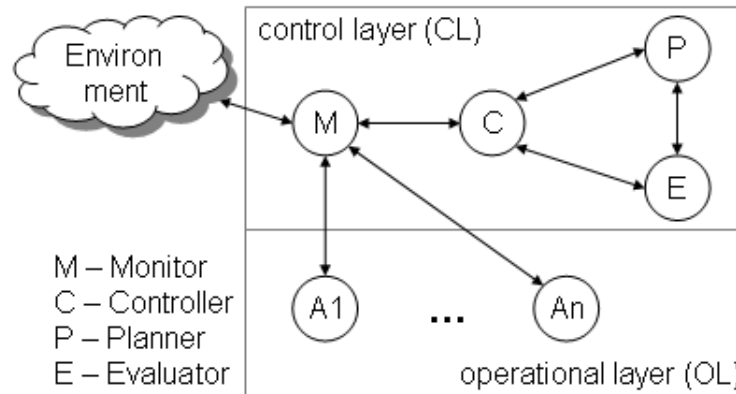


Figure 7.22: Self-configuring MAS: 2-layered architecture

Agent	Data Stored	Actions Performed	Communication
<i>Monitor</i>	Read-only access to <i>problemDef</i> , <i>D</i> , <i>G</i> .	Monitors (listens) OL and the environment; notifies <i>Controller</i> about triggering changes; propagates the new plan to OL agents.	Environment, OL agents, <i>Controller</i> .
<i>Controller</i>	<i>problemDef</i> , <i>actionLog</i> , <i>removeActionLog</i> , <i>G</i> , <i>D</i> .	Follows the reconfiguration mechanism, exploiting <i>Planner</i> (to initiate replanning) and <i>Evaluator</i> (to evaluate loads); updates stored data structures; notifies <i>Monitor</i> about plan changes.	<i>Monitor</i> , <i>Planner</i> , <i>Evaluator</i> .
<i>Planner</i>	Domain definition; read-only access to <i>problemDef</i> .	Performs planning; tunes <i>problemDef</i> in accordance with <i>Evaluator's</i> results.	<i>Controller</i> , <i>Evaluator</i> .
<i>Evaluator</i>	Action costs and load bounds for each OL agent.	Follows the evaluation procedure; evaluates OL agents load.	<i>Controller</i> , <i>Planner</i> .

Table 7.6: Self-configuring MAS: agents of the control layer

The presented self-configuration approach can be applied both to socio-technical systems, and to the multi-agent systems, which comprise only software agents. However, the application of the approach to the former type of systems can be much more beneficial, as dynamicity and the self-interested rational behavior are among the STS intrinsic properties.

The proposed reconfiguration mechanism is limited in that it supports only four types of triggering events, namely, the situations when a new actor enters the system, or the existing one leaves, when a new system goal is introduced, or one of the old ones is satisfied. However, the formalization could be quite easily extended to support the changes in the actors' capabilities and commitments, failures when achieving goals, etc. This is possible due to the flexibility of the PDDL representation [46] of the problem and the planning domain.

7.4 From organizational to instance level design

One of the critical issue in designing a socio-technical system is moving from an organization structure to its actual implementation, where roles and activities have to be assigned to specific agents (either humans or software components). Making a clear separation between the organizational and the instance level of an STS allows the designer to verify various constraints and properties that cannot be expressed at organizational level, but are relevant only at instance level (e.g. conflict of interests). In this section, we apply our planning-based framework to the domain of socio-technical system design, considering both organizational and instance level perspectives.

Most of the existing modelling frameworks remain silent on how organizational and instance level models are related, a rare counter example is [128], where the relation between the two is discussed and the formal definition of model instantiation axioms is given. To a certain extent, similar problems are considered in the area of requirements simulation [107], where the focus is usually on the validation of system specifications.

The separation of modelling and analysis levels is important, as there are constraints that cannot be verified at the organizational level, but are related to an instantiated socio-technical model. One example of such constraint is avoiding the *conflict of interests*, which concrete realization consists in not allowing an agent to play two certain roles at a time. Following the ideas presented in [128], our goal is to provide a Tropos-based framework for modelling and analyzing socio-technical systems at both organizational and instance levels. This framework will support model transformation and allow for formal specification of a vast range of constraints applied to roles as well as to agents playing these roles.

In this work, we adopt the planning based approach presented in Chapter 4 for generating both organizational and instance level designs, and show how these two are related, how an input setting at both levels is formalized and how the various constraints are encoded. In the following, we first present a motivating case study, and then explain the approach.

7.4.1 A motivating case study

The case study we will use in the paper, originates from SERENITY EU project⁸. The focus of the case study is on banking, namely, on how loans are provided by banks to their customers. A detailed description of the case study is given in the corresponding project deliverable [110], while in the paper, for the sake of clarity and due to space limitations, we present only a fragment of it.

The main actors of the banking scenario are *Customer* and *Bank*. A customer is willing to obtain a loan (e.g. in order to buy a house), and a bank is capable of providing loans. In Tropos terminology, a customer, which is a *role* that can be played by various human *agents*, depends on a bank for the *goal* of obtaining a loan. Although a bank has a complex organizational structure, we consider only three roles within the bank played by its employees, namely, *Bank Manager*, *Pre-processing Clerk* and *Post-processing Clerk*. A manager can, in principle, be involved in all the steps of the loan origination process, but usually he participates to the activities in which his explicit approval or supervision is required.

The key difference between a pre-processing and a post-processing clerk is in the experience needed to accomplish the duties associated with these roles. Namely, a pre-processing clerk covers the early steps of the loan origination process which require less skills and responsibility, while a post-processing clerk is an experienced bank employee, who is responsible for performing all the banking transactions for the customer, including checking customer credit worthiness. For the latter check, a post-processing clerk can contact a *Consultant* of the *Credit Bureau*, which is an external actor with respect to the bank. Credit Bureau is “a third-party business partner of a financial institution that processes, stores and safeguards credit information of physical and industrial companies” [110].

Some additional information about the customer can be requested from various state institutions, e.g. a bank might be interested in checking a customer criminal record before providing a loan, so the relevant authority will be consulted for this purpose. In the banking scenario, all such sources of information are collectively represented as an agent called *Government*.

The fragment of the case study we consider in this paper, contains only human and organizational actors, however, this does not mean that our approach does not support the analysis of systems which include software actors as well. The original scenario includes a technical component, *Internal Computer System*, which is capable of storing data, performing different calculations (e.g. of a loan price), generating contract templates, etc. Also, the full version of the case study contains much more details on how various activities are carried out, what resources are used, etc. Due to space limitations, in this paper the above mentioned details are omitted.

Two important questions should be addressed when analyzing the presented scenario. The first question, which was considered in details in Chapters 4 and 5, is about the optimal, or good-enough organizational structure capable of achieving the stakeholders’ goals. The second question to ask is the following: Are there any additional requirements and constraints related to an *instance* of an organizational model, and if so, how concretely

⁸<http://www.serenity-project.org/>

should they be taken into account? To answer this, a concrete instantiation mapping and a way to formally specify the constraints should be defined.

In the above fragment of the banking scenario, despite its simplicity, a number of instance level constraints can be identified. Take, for instance, the activity of checking customer credit worthiness. In order to increase the reliability of the whole process, this check should be performed by two different persons: a bank clerk, performing an internal check, and a consultant at the credit bureau, performing an additional check based on the information which might not be available to the bank. However, this constraint can be violated by a bank clerk who is also a part-time consultant at the credit bureau, and it would be useful to have a support for identifying and fixing such violations. Or, alternatively, it would be beneficial to have a supporting tool for constructing a good-enough (with respect to some predefined constraints) instantiation of an organizational model. This latter challenge is addressed in this work.

7.4.2 General schema of the approach

Figure 7.23 presents the general schema of the design approach we propose, which includes the following steps.

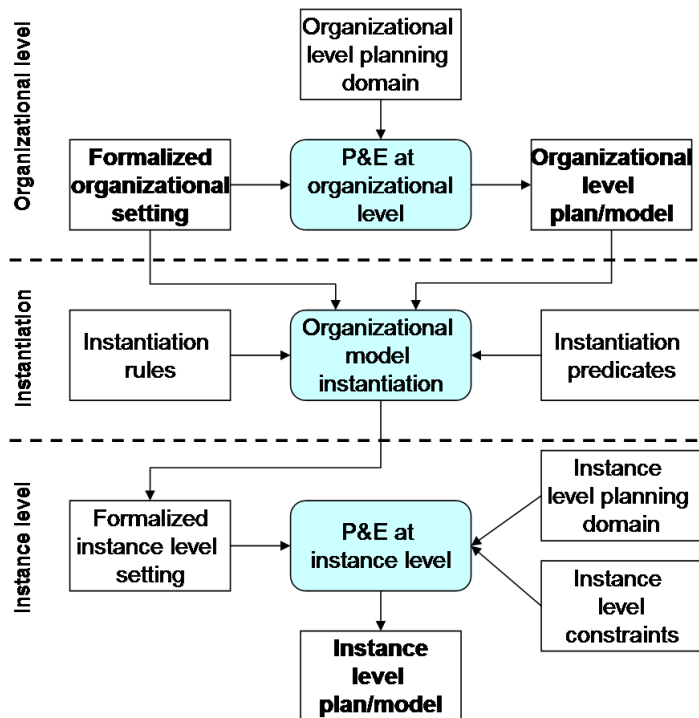


Figure 7.23: General schema of the approach

Planning and evaluation at the organizational level. This step takes as an input the formalized initial organizational setting in terms of actors and goals, and the specification of the organizational level planning domain. It adopts planning techniques for constructing organizational models, and uses the planning and evaluation

procedure presented in Chapters 4 and 5, which allows obtaining a model which is optimal or good-enough with respect to a number of user defined criteria.

Organizational model instantiation. This step takes as an input the organizational level plan produced on a previous step, formalized initial organizational setting, instantiation predicates which specify the domain agents and the roles they play, and a set of instantiation rules. At this step, a socio-technical model is instantiated in accordance with the above rules. An output of this step is a formalized instantiated organizational setting.

Planning and evaluation at the instance level. This step is analogous to the first one, and takes as an input the formalized organizational setting produced on the previous step, as well as the instance level planning domain, which includes a number of instance level constraints discussed in Section 7.4.4.

To summarize, the above process provides an automation support for a number of subsequent modelling and analysis activities, from building a high level organizational model, to analyzing instance level model constraints (e.g. role or assignment conflicts). The two main artifacts produced by the process are an organizational and a corresponding instantiated model, both compliant by construction with a set of predefined properties and constraints. Obviously, the design process cannot be fully automated, so a human designer remains in the loop for approving or rejecting proposed alternatives, providing feedback and making final design decisions.

7.4.3 Planning at the organizational level: example

Here the difference in the planning domain definition from the one presented in Chapter 4 is that an **actor** is specialized into an **agent** or a **role**. The notion of actor itself is used to specify an input setting at the organizational level, as at this level there is no need to distinguish between roles and agents. E.g. in the banking scenario, at the organizational level we treat the agent *government* (assuming that the scope of the case study is limited to one country) and the role *consultant* in the same way: we apply the same predicates to describe the properties of both and the same actions to construct a plan which involves both. This is so because at the organizational level the aim is to understand the relations between a “generic” consultant and the government, while the actual number of consultants is not yet relevant at this level.

Unlike actors, **goals** are used in both organizational and instance level formalization. As explained later in Section 7.4.4, a key idea of goal instantiation is replacing an organizational level goal, associated with a role, with a number of goal instances, each associated with an agent playing this role. So, e.g. if in the instance level model there are two customers, Carol and Cid, then there will be two instances of goal *launch loan origination process*, one associated with “Carol playing the role of a customer” and the other with “Cid playing the role of a customer”.

In Figure 7.24 the formalization of the fragment of the banking scenario at the organizational level is shown. To understand the details of formalization, the reader can refer to the diagram in Figure 7.26 (some of the relations present in this diagram will be explained

later, in Section 7.4.4). In this scenario, the goal of *launching the loan origination process* is AND-decomposed into *receiving a customer* and *managing the loan origination process*. The latter goal is further decomposed into *checking customer's trustworthiness*, *preparing a proposal* and *signing a contract for a loan*. Checking customer's trustworthiness consists in performing two checks, an internal one, which can be done by a post-processing clerk, and an external one, which can be done by a consultant of the external Credit Bureau. To perform a customer check, a consultant should use two sources of information: the financial data on customer's credit trustworthiness, which is collected and stored internally by the Credit Bureau, and the data on customer's "general" trustworthiness, which is connected to a customer's criminal record, and can be obtained via an inquiry to the appropriate governmental institution.

```

LaunchLoanOrProc ReceiveCust ManageLoanOr CheckCust
  PrepProposal SignContract IntCheck ExtCheck CheckCrimR
  CheckCredTr StoreCrimR – goal
Customer Manager PreClerk PostClerk Consultant
  Government – actor
Customer Manager PreClerk PostClerk Consultant – role
Government – agent

(and_subgoal2 LaunchLoanOrProc ReceiveCust ManageLoanOr)
(and_subgoal3 ManageLoanOr
  CheckCust PrepProposal SignContract)
(and_subgoal2 CheckCust IntCheck ExtCheck)
(and_subgoal2 ExtCheck CheckCrimR CheckCredTr)
(means_end StoreCrimR CheckCrimR)
(can_depend_on Customer Manager)
(can_depend_on_g Manager PreClerk ReceiveCust)
(can_depend_on_g Manager PostClerk ManageLoanOr)
(can_depend_on Consultant Government)
(can_depend_on_g PostClerk Consultant ExtCheck)
(can_satisfy PreClerk ReceiveCust)
(can_satisfy PostClerk IntCheck)
(can_satisfy PostClerk PrepProposal)
(can_satisfy PostClerk SignContract)
(can_satisfy Consultant CheckCredTr)
(can_satisfy Government StoreCrimR)
(wants Customer LaunchLoanOrProc)

```

Figure 7.24: Banking scenario: formalization

Also, in the formalization there is a means-end link between *store criminal record* and *check customer's criminal record*, which means that the satisfaction of the former goal implies the satisfaction of the latter. Of course, the decomposition tree of the goal *check customer's criminal record* is more complicated, namely, a consultant should formulate an inquiry to the government, which is then processed and the answer is sent back, but we omit these details for the sake of simplicity.

In addition, in Figure 7.24, possible dependencies among actors and actor capabilities are specified, and the *wants* predicate represents the motivation beyond the whole scenario: a customer wants a loan origination process to be launched. Note, that no *type* and *conflict* predicates are present in the formalization of this scenario.

Figure 7.25 presents the plan obtained for the banking scenario formalized in Figure 7.24. In Figure 7.26 the corresponding organizational model is depicted. In this specific case, the model is a straightforward mapping of the initial organizational setting to Tropos modelling primitives. Obviously, in more complex cases, the initial organizational setting allows for a number of alternative designs.

DELEGATES	Customer	Manager	LaunchLoanOrProc
AND_DECOMPOSES₂	Manager	LaunchLoanOrProc	ReceiveCust
	Manager	ManageLoanOr	
DELEGATES	Manager	PreClerk	ReceiveCust
SATISFIES	PreClerk	ReceiveCust	
DELEGATES	Manager	PostClerk	ManageLoanOr
AND_DECOMPOSES₃	PostClerk	ManageLoanOr	CheckCust
	PrepProposal	SignContract	
AND_DECOMPOSES₂	PostClerk	CheckCust	IntCheck
	ExtCheck		
SATISFIES	PostClerk	IntCheck	
DELEGATES	PostClerk	Consultant	ExtCheck
AND_DECOMPOSES₂	Consultant	ExtCheck	CheckCrimR
	CheckCredTr		
SATISFIES	Consultant	CheckCredTr	
SATISFIES	Government	StoreCrimR	
SATISFIES	PostClerk	PrepProposal	
SATISFIES	PostClerk	SignContract	

Figure 7.25: Banking scenario: an organizational level plan

7.4.4 Organizational model instantiation

The key idea behind model instantiation, similarly to [128], is the following: *a role has as many instances as there are agents playing this role, and a goal has as many instances as there are agents who want this goal to be satisfied*. As it was discussed earlier, certain constraints can be expressed and analyzed only at the instance level, where a socio-technical model reflects the actual dependencies among agents playing roles and goal assignments to these agents. The source of instance level constraints can be legal regulations (e.g. two activities can not be performed by the same agent) or optimization concerns (e.g. overlapping assignments of goals to agents should be avoided).

Instantiation may start either from an organizational level plan, or from a formalized initial organizational setting. In the former case, the resulting instance level plan will be an instance of an organizational level plan, while in the latter case the search space for an instance level plan is bigger due to those relations and capabilities that are “potentially” present at the organizational level (e.g. OR-decompositions of goals, actor capabilities which are not used in the organizational level model). In our approach, both cases are supported.

To formalize an instantiated organizational setting, a set of predicates presented in Table 7.7 is used.

Instance level predicates (“Agent-Role Properties and Relations” section in Table 7.7) describe agents, roles and their relations with goals, and are defined analogously to the

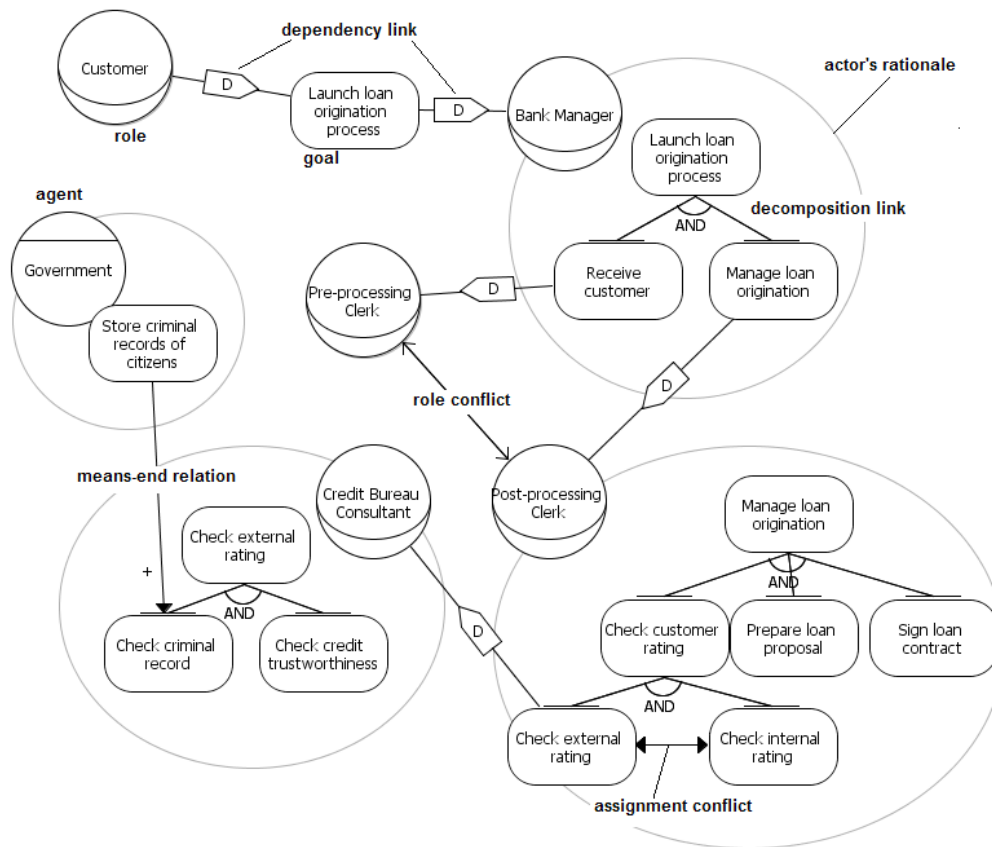


Figure 7.26: Banking scenario: organizational level model

predicates in the basic set of Section 4.1. In all the instance level predicates an agent is coupled with the role it plays. E.g. $\text{can_satisfy}(a: \text{agent}, r: \text{role}, g: \text{goal})$ should be read as “agent a when playing role r is capable of satisfying goal g ”.

Instantiation predicates are the following two: $\text{plays}(a: \text{agent}, r: \text{role})$ representing the situation in which an agent is actually playing a role, and $\text{can_play}(a: \text{agent}, r: \text{role})$ meaning that an agent can potentially play a role, i.e. has all the necessary abilities for it. Note that for the instantiation procedure to be consistent, we do not allow $\text{can_play}(a, r)$ predicates for those roles r for which $\text{wants}(r, g)$ is in the organizational level setting description for some goal g .

Instantiation constraints represent two situations that should be avoided in a resulting model, namely, role and goal assignment conflicts. The former is encoded by $\text{role_conflict_a}(r_1, r_2: \text{role}, a: \text{agent})$ predicate, meaning that agent a cannot play roles r_1 and r_2 at the same time. The “generic” version of this predicate, in which an agent is omitted, means that no agent is allowed to play the specified roles at the same time. The second type of constraint, goal assignment conflict, encoded by $\text{assignment_conflict_a}(g_1, g_2: \text{goal}, a: \text{agent})$ predicate, represents the fact that goals g_1 and g_2 cannot be both satisfied by the same agent a . Again, the “generic” version of the predicate has the same meaning for any agent of the domain.

Instantiation Predicates and Constraints
<code>can_play(a : agent, r : role)</code> <code>plays(a : agent, r : role)</code> <code>assignment_conflict(g₁, g₂ : goal)</code> <code>assignment_conflict_a(g₁, g₂ : goal, a : agent)</code> <code>role_conflict(r₁, r₂ : role)</code> <code>role_conflict_a(r₁, r₂ : role, a : agent)</code>
Agent-Role Properties and Relations
<code>can_satisfy(a : agent, r : role, g : goal)</code> <code>can_satisfy_gt(a : agent, r : role, gt : gtype)</code> <code>wants(a : agent, r : role, g : goal)</code> <code>can_depend_on(a₁ : agent, r₁ : role, a₂ : agent, r₂ : role)</code> <code>can_depend_on_gt(a₁ : agent, r₁ : role, a₂ : agent, r₂ : role, gt : gtype)</code> <code>can_depend_on_g(a₁ : agent, r₁ : role, a₂ : agent, r₂ : role, g : goal)</code>

Table 7.7: Formalizing the instantiated setting: predicates

As in the case of an organizational level formalization, certain rules, or *axioms*, should hold:

1. possible agent dependency on a goal type implies the same dependency on all goals of this type:

$$\text{can_depend_on_gt}(a_1, r_1, a_2, r_2, \text{gt}), a_1, a_2 : \text{agent}, r_1, r_2 : \text{role}, \text{gt} : \text{gtype} \rightarrow \forall g : \text{goal.type}(g, \text{gt}) \text{can_depend_on_g}(a_1, r_1, a_2, r_2, g);$$
2. agent capability to satisfy a goal type implies the capability to satisfy all goals of this type:

$$\text{can_satisfy_gt}(a : \text{agent}, r : \text{role}, \text{gt} : \text{gtype}) \rightarrow \forall g : \text{goal.type}(g, \text{gt}) \text{can_satisfy_g}(a, r, g);$$
3. generic assignment conflict implies assignment conflict for any agent of the domain:

$$\text{assignment_conflict}(g_1, g_2 : \text{goal}) \rightarrow \forall a : \text{agent} \text{assignment_conflict_a}(g_1, g_2, a);$$
4. goal assignment conflict is symmetric:

$$\text{assignment_conflict_a}(g_1, g_2 : \text{goal}, a : \text{agent}) \rightarrow \text{assignment_conflict_a}(g_2, g_1, a);$$
5. generic role conflict implies role conflict for any agent of the domain:

$$\text{role_conflict}(r_1, r_2 : \text{role}) \rightarrow \forall a : \text{agent} \text{role_conflict_a}(r_1, r_2, a);$$
6. role conflict is symmetric:

$$\text{role_conflict_a}(r_1, r_2 : \text{role}, a : \text{agent}) \rightarrow \text{role_conflict_a}(r_2, r_1, a).$$

Let us consider the instantiation predicates and constraints for the banking scenario listed in Figure ???. Seven new agents are introduced, *Cid* and *Carol*, the two customers, *Mike* and *Pat*, which can play the role of a bank manager and pre-processing clerk, respectively, *Chris*, who can play the role of a consultant, *Peter*, who can be both a consultant or a post-processing clerk, and *Paul*, who is capable of playing the roles of both pre- and post-processing clerk. Two examples of instance level constraints, represented also graphically in Figure 7.26, are the following. The first one is an assignment conflict between *checking customer's trustworthiness internally* in the bank, and *externally* via

an inquiry to the Credit Bureau. Namely, these checks cannot be performed by the same person, which may happen if a bank employee works as a part-time consultant, like Peter can do in our example. The second constraint refers to the conflict between the roles of pre- and post-processing clerk, namely, an agent is allowed to play only one of the roles at the same time. This can be, in principle, violated by Paul in our example.

Peter Paul Pat Cid Carol Mike Chris – agent (plays Cid Customer) (plays Carol Customer) (can_play Chris Consultant) (can_play Peter Consultant) (can_play Peter PostClerk) (can_play Paul PostClerk) (can_play Paul PreClerk) (can_play Pat PreClerk) (assignment_conflict CheckCredTr IntCheck) (role_conflict PreClerk PostClerk)

Figure 7.27: Banking scenario: instantiation input

An organizational model is instantiated following the rules, detailed and illustrated on the banking scenario in Table 7.8. These rules specify how to form an instance level planning problem (*IP*) starting from a planning problem at the organizational level (*OP*).

7.4.5 Planning at the instance level

A plan constructed for the instantiated organizational setting consists of the following actions.

TAKES_ON_ROLE($a : agent, r : role$): agent a takes on all the rights and responsibilities associated with role r ;

SATISFIES($a : agent, r : role, g : goal$): agent a playing role r satisfies goal g ;

AND/OR_DECOMPOSES $_n(a : agent, r : role, g, g_1, \dots, g_n : goal)$: agent a playing role r decomposes goal g into its n AND/OR-subgoals;

DELEGATES ($a_1 : agent, r_1 : role, a_2 : agent, r_2 : role, g : goal$): agent a_1 playing role r_1 depends for goal g on agent a_2 playing role r_2 .

Role and assignment conflicts are taken into account when specifying the preconditions of actions **TAKES_ON_ROLE** and **SATISFIES** as shown in Table 7.9. In the table, **pr_satisfies**($a : agent, r : role, g : goal$) is a “trace predicate”, which becomes true after an action **SATISFIES**(a, r, g) is added to a plan.

In the instantiation of the banking scenario, a possible solution generated by the planner contains the actions presented in Figure 7.28.

The remaining plan actions are the instances of the class level plan presented in Figure 7.25. In this instance level model, Cid is served by Peter playing the role of a post-processing clerk and Chris playing the role of a consultant, and Carol is served by Paul

Step	Description and examples
1. Agents and roles	Add roles and agents from <i>OP</i> , add the new agents specified in the instantiation input; add <code>role_conflict_a(r₁, r₂, a)</code> predicates specified in the instantiation constraints; Example Customer Manager PreClerk PostClerk Consultant – role Government Peter Paul Pat Cid Carol Mike Chris – agent (role_conflict PreClerk PostClerk)
2. Agent-role relations	Add all <code>plays(a : agent, r : role)</code> and <code>can_play(a : agent, r : role)</code> predicates specified in the instantiation input; a “fake” role <i>norole</i> is introduced and <code>plays(a : agent, norole)</code> is added for all agents <i>a</i> for which no role is specified Example (plays Cid Customer) (can_play Chris Consultant) (plays Government norole)
3. Goals and agent desires	For each goal <i>g</i> in <i>OP</i> add a goal with the name <i>r-a-g</i> for each (<i>r</i> : role, <i>a</i> : agent) pair such that <i>OP</i> contains <code>wants(r, g)</code> , and <i>IP</i> contains either <code>play(a, r)</code> or <code>can_play(a, r)</code> ; then, instantiate <code>wants(a, g)</code> : <code>wants(r, g), r : role → ∀a : agent, r-a-g : goal wants(a, r, r-a-g)</code> <code>wants(a, g), a : agent → wants(a, norole, g)</code> Example Customer-Cid-LaunchLoanOrProc Customer-Carol-LaunchLoanOrProc – goal (wants Cid Customer Customer-Cid-LaunchLoanOrProc) (wants Carol Customer Customer-Carol-LaunchLoanOrProc)
4. Goal decomposition	Instantiate <code>and/or_subgoal_n(g, g₁, ..., g_n)</code> by adding the corresponding predicates for each goal instance <i>*-g</i> of <i>g</i> ; if any of <i>g_i</i> is not yet instantiated, add an instance <i>r-a-g_i</i> for each <i>r-a-g</i> . Example Customer-Cid-ReceiveCust Customer-Cid-ManageLoanOr – goal (and_subgoal ₂ Customer-Cid-LaunchLoanOrProc Customer-Cid-ReceiveCust Customer-Cid-ManageLoanOr)
5. Goal conflicts	For each <code>conflict(g₁, g₂)</code> for all (<i>r</i> : role, <i>a</i> : agent) such that <i>r-a-g₁</i> is an instance of <i>g₁</i> and <i>g₂</i> is not yet instantiated, instantiate <i>g₂</i> with <i>r-a-g₂</i> ; then, instantiate the relation as follows <code>conflict(g₁, g₂), g₁, g₂ : goal → ∀r : role, a : agent. ∃r-a-g₁, r-a-g₂ : goal conflict(r-a-g₁, r-a-g₂)</code>
6. Means-end	For each <code>means_end(g_{means}, g_{end})</code> for all (<i>r</i> : role, <i>a</i> : agent) such that <i>r-a-g_{end}</i> is an instance of <i>g_{end}</i> , and <i>g_{means}</i> is not yet instantiated, instantiate <i>g_{means}</i> with <i>r-a-g_{means}</i> ; then, instantiate the relation: <code>means_end(g_{means}, g_{end}) → ∀r : role, a : agent. ∃r-a-g_{means}, r-a-g_{end} means_end(r-a-g_{means}, r-a-g_{end})</code> Example Customer-Carol-StoreCrimR – goal (means_end Customer-Carol-StoreCrimR Customer-Carol-CheckCrimR)
7. Agent capabilities	Instantiate <code>can_satisfy(a, g)</code> as follows <code>can_satisfy(r, g), r : role → ∀a : agent, *-g : goal can_satisfy(a, r, *-g)</code> <code>can_satisfy(a, g) a : agent → can_satisfy(a, norole, *-g)</code> ; Example (can_satisfy Chris Consultant Customer-Cid-CheckCredTr) (can_satisfy Government norole Customer-Cid-StoreCrimR)
8. Dependencies	Instantiate <code>can_depend_g(a₁, a₂, g)</code> as follows <code>can_depend_g(r₁, r₂, g), r₁, r₂ : role → ∀a₁, a₂ : agent, *-g : goal can_depend_g(a₁, r₁, a₂, r₂, *-g)</code> <code>can_depend_g(a₁, r₂, g), a₁ : agent, r₂ : role →</code> <code>∀a₂ : agent, *-g : goal can_depend_g(a₁, norole, a₂, r₂, *-g)</code> <code>can_depend_g(r₁, a₂, g), r₁ : role, a₂ : agent →</code> <code>∀a₁ : agent, *-g : goal can_depend_g(a₁, r₁, a₂, norole, *-g)</code> <code>can_depend_g(a₁, a₂, g), a₁, a₂ : agent → ∀*-g : goal can_depend_g(a₁, norole, a₂, norole, *-g)</code> ; Example (can_depend_on_g Paul PostClerk Chris Consultant Customer-Cid-ExtCheck) (can_depend_on_g Paul PostClerk Peter Consultant Customer-Cid-ExtCheck)
9. Assignment conflicts	Instantiate <code>assignment_conflict_a</code> as follows <code>assignment_conflict_a(g₁, g₂, a) →</code> <code>∀r : role. ∃r-a-g₁, r-a-g₂ : goal assignment_conflict_a(r-a-g₁, r-a-g₂, a)</code> ; Example assignment_conflict_a(Customer-Cid-CheckCredTr, Customer-Cid-IntCheck, Peter) assignment_conflict_a(Customer-Carol-CheckCredTr, Customer-Carol-IntCheck, Peter)

Table 7.8: Organizational model instantiation: steps and examples

TAKES_ON_ROLE ($a : agent, r : role$)
precondition: $can_play(a, r) \wedge$ $\forall r' : role \neg(play(a, r') \wedge role_conflict(r, r', a))$
effect: $plays(a, r)$
SATISFIES ($a : agent, r : role, g : goal$)
precondition: $plays(a, r, g) \wedge$ $wants(a, r, g) \wedge$ $can_satisfy(a, r, g) \wedge$ $\forall g' : goal \neg(conflict(g, g') \wedge satisfied(g')) \wedge$ $\forall g' : goal \neg(means_ends(g', g) \wedge satisfied(g')) \wedge$ $\forall g' : goal, r' : role$ $\neg(pr_satisfies(a, r', g') \wedge assignment_conflict(g, g', a))$
effect: $satisfied(g) \wedge$ $\neg wants(a, r, g)$

Table 7.9: Role assignment and satisfaction actions at the instance level

TAKES_ON_ROLE Mike Manager
TAKES_ON_ROLE Pat PreClerk
TAKES_ON_ROLE Paul PostClerk
TAKES_ON_ROLE Peter Consultant
TAKES_ON_ROLE Peter PostClerk
TAKES_ON_ROLE Chris Consultant

Figure 7.28: Banking scenario: (a fragment of) an instance level plan

playing the role of a post-processing clerk and Peter playing the role of a consultant. By construction, this model satisfies all the instance level constraints.

Both organizational and instance level planning domains are implemented using PDDL 2.2 specification language [46] and LPG-td planner [90]. The instantiation procedure, which steps are explained in Section 7.4.4, is implemented in Perl scripting language.

.....

7.5 Other application domains

AmiCriM?

Chapter 8

Conclusion – TO FINISH

Bibliography

- [1] Specware: Automated Software Development System – <http://www.specware.org/>.
- [2] JADE: Java Agent DEvelopment Framework website — <http://jade.tilab.com/>.
- [3] FIPA: Foundation for Intelligent Physical Agents — <http://www.fipa.org/>.
- [4] Susanne Albers, Stefan Eilts, Eyal Even-Dar, Yishay Mansour, and Liam Roditty. On Nash Equilibria for a Network Creation Game. In *SODA '06*, pages 89–98, 2006.
- [5] AMICE Consortium. *Open System Architecture for CIM*. Springer-Verlag New York, Inc., 1993.
- [6] John S. Anderson and Stephen Fickas. A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. In *IWSSD'89*, pages 177–184, 1989.
- [7] Elliot Anshelevich, Anirban Dasgupta, Eva Tardos, and Tom Wexler. Near-Optimal Network Design with Selfish Agents. In *STOC'03*, pages 511–520, 2003.
- [8] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 8–12, New York, NY, USA, 2004. ACM Press.
- [9] Yudistira Asnar, Volha Bryl, and Paolo Giorgini. Using risk analysis to evaluate design alternatives. In Lin Padgham and Franco Zambonelli, editors, *AOSE*, volume 4405 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2006.
- [10] M. Bryce & Associates. PRIDE-EEM Enterprise Engineering Methodology. <http://www.phmainstreet.com/mba/pride/eemeth.htm>, 2006.
- [11] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
- [12] J. Baier, F. Bacchus, and S. McIlraith. A Heuristic Search Approach to Planning with Temporally Extended Preferences. In *IJCAI'07*, pages 1808–1815, 2007.
- [13] R. Balzer, Jr. Cheatham T.E., and C. Green. Software technology in the 1990's: Using a new paradigm. *Computer*, 16(11):39–45, 1983.

- [14] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. 11(3):207–230, 2001.
- [15] Carole Bernon, Marie Pierre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. Adelfe: A methodology for adaptive multi-agent systems engineering. In *ESAW*, pages 156–169, 2002.
- [16] Peter Bernus and Laszlo Nemes. A framework to define a generic enterprise reference architecture and methodology. In *IFAC/IFIP Task Force on Architectures for Enterprise Integration*, pages 179–191, 1994.
- [17] Stefan Biffl, Aybüke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher. *Value-Based Software Engineering*. Springer-Verlag New York, Inc., 2005.
- [18] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, Mills, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3), 2002.
- [19] Barry Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.
- [20] Barry Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.
- [21] Barry Boehm. A view of 20th and 21st century software engineering. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 12–29. ACM, 2006.
- [22] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [23] Paolo Bresciani and Paolo Donzelli. A practical agent-based approach to requirements engineering for socio-technical systems. In *AOIS'03*, pages 158–173, 2003.
- [24] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAA-MAS*, 8(3):203–236, 2004.
- [25] Paolo Bresciani, Paolo Giorgini, Haralambos Mouratidis, and Gordon Manson. Multi-agent Systems and Security Requirements Analysis. *Software Engineering for Multi-Agent Systems II*, pages 35–48, 2004.
- [26] Volha Bryl and Paolo Giorgini. Self-Configuring Socio-Technical Systems: Redesign at Runtime. In *SOAS'06*, 2006.
- [27] Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing Cooperative IS: Exploring and Evaluating Alternatives. In *CoopIS'06*, pages 533–550, 2006.
- [28] Volha Bryl, Paolo Giorgini, and John Mylopoulos. Supporting Requirements Analysis in Tropos: a Planning-Based Approach. In *PRIMA'07*, 2007.

- [29] Volha Bryl, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Designing Security Requirements Models Through Planning. In *CAiSE'06*, pages 33–47, 2006.
- [30] Volha Bryl, Paola Mello, Marco Montali, Paolo Torroni, and Nicola Zannone. B-Tropos: Agent-oriented requirements engineering meets computational logic for declarative business process modeling and verification. 2007.
- [31] Giovanni Caire, Wim Coulier, Francisco J Garijo, Jorge Gomez, Juan Pavon, Francisco Leal, Paulo Chainho, Paul E. Kearney, Jamie Stark, Richard Evans, and Philippe Massonet. Agent oriented analysis using Message/UML. In *Agent-Oriented Software Engineering II, Second International Workshop*, volume 2222, pages 119–135. Springer, 2001.
- [32] James Caldwell. Moving Proofs-as-Programs into Practice. pages 10–17, 1997.
- [33] L. Castillo, J. Fdez-Olivares, and A. Gonzalez. Integrating Hierarchical and Conditional Planning Techniques into a Software Design Process for Automated Manufacturing. In *ICAPS'03*, pages 28–39, 2003.
- [34] L. Cernuzzi and F. Zambonelli. Dealing with adaptive multi-agent organizations in the gaia methodology. In *Workshop on Agent-Oriented Software Engineering (AOSE 2005)*, pages 217–228, 2005.
- [35] Betty H.C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 285–303, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [36] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
- [37] John Clarke, Jose Javier Dolado, Mark Harman, Bryan Jones, Mary Lumkin, Brian Mitchell, Kearton Rees, and Marc Roper. Reformulating software engineering as a search problem. *IEE Proceedings on Software*, 150(3):161–175, June 2003.
- [38] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., 1986.
- [39] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.
- [40] Rajdeep K. Dash, Nicholas R. Jennings, and David C. Parkes. Computational Mechanism Design: A Call to Arms. *IEEE Intelligent Systems*, 18(6):40–47, 2003.
- [41] Marina Davidson and Max Garagnani. Pre-processing Planning Domains Containing Language Axioms. In *PlanSIG'02*, pages 23–34, 2002.
- [42] Aldo de Moora and Hans Weigandb. Formalizing the evolution of virtual communities. *Inornation Systems*, 32(2):223–247, 2007.

- [43] Scott A. DeLoach and Eric Matson. An organizational model for designing adaptive multiagent systems. In *Proceedings of AAAI-04 Workshop on Agent Organizations*, pages 66–73, 2004.
- [44] Virginia Dignum. *A Model for Organizational Interaction, based on Agents, founded in Logic*. PhD thesis, University of Utrecht, 2003.
- [45] Virginia Dignum, Liz Sonenberg, and Frank Dignum. Towards dynamic reorganization of agent societies. In *Workshop on Coordination in Emergent Agent Societies*, 2004.
- [46] Stefan Edelkamp and Jorg Hoffmann. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, University of Freiburg, 2004.
- [47] Thomas Ellman. Specification and Synthesis of Hybrid Automata for Physics-Based Animation. In *ASE'03*, pages 80–93, 2003.
- [48] F.E. Emery. Characteristics of socio-technical systems. *London: Tavistock*, 1959.
- [49] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS*, pages 128–135, 1998.
- [50] Anthony Finkelstein, Mark Harman, S. Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. "fairness analysis" in requirements assignments. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE '08)*, 2008.
- [51] Ian Foster. What is the Grid? A Three Point Checklist, 2002.
- [52] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20:61–124, 2003.
- [53] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE'07: Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [54] Xavier Franch. On the Quantitative Analysis of Agent-Oriented Models. In *CAiSE'06*, pages 495–509, 2006.
- [55] Jr. Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [56] Ariel Fuxman, Paolo Giorgini, Manuel Kolp, and John Mylopoulos. Information systems as social structures. pages 10–21, 2001.
- [57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [58] G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the Impact of Trust and Distrust in Agent Networks. In *AOIS'01*, pages 45–58, 2001.

- [59] Massimiliano Garagnani. A Correct Algorithm for Efficient Planning with Preprocessed Domain Axioms. *Research and Development in Intelligent Systems XVII*, pages 363–374, 2000.
- [60] Maddalena Garzetti, Paolo Giorgini, John Mylopoulos, and Fabrizio Sannicolò. Applying Tropos Methodology to a real case study: Complexity and Criticality Analysis. In *WOA'02*, pages 7–13, 2002.
- [61] B. Cenk Gazen and Craig A. Knoblock. Combining the Expressivity of UCPOP with the Efficiency of Graphplan. In *ECP'97*, pages 221–233, 1997.
- [62] Alfonso Gerevini and Derek Long. Plan Constraints and Preferences in PDDL3. Technical Report RT 2005-08-47, University of Brescia, Italy, 2005.
- [63] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – The Planning Domain Definition Language. In *AIPS'98*, 1998.
- [64] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. ConGolog, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
- [65] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with Goal Models. In *ER'02*, pages 167–181, 2002.
- [66] Paolo Giorgini, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Modeling Security Requirements Through Ownership, Permission and Delegation. In *RE'05*, pages 167–176, 2005.
- [67] Andreas Gregoriades, Jae-Eun Shin, and Alistair Sutcliffe. Human-centred requirements engineering. In *RE'04*, pages 154–163, 2004.
- [68] Daniel Gross and Eric S. K. Yu. From Non-Functional Requirements to Design through Patterns. *Requirements Engineering*, 6(1):18–36, 2001.
- [69] Giancarlo Guizzardi, Gerd Wagner, Nicola Guarino, and Marten van Sinderen. An Ontologically Well-Founded Profile for UML Conceptual Models. volume 3084, pages 112–126, 2004.
- [70] Jon G. Hall and Andres Silva. A requirements-based framework for the analysis of socio-technical system behaviour. In *REFSQ'03*, pages 117–120, 2003.
- [71] Brian Henderson-Sellers and Paolo Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group Publishing, 2005.
- [72] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.
- [73] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19(4):281–316, 2004.

- [74] Jomi Fred Hubner, Jaime Simao Sichman, and Olivier Boissier. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In *SBIA'02: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence*, pages 118–128, London, UK, 2002. Springer-Verlag.
- [75] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Using the moise+ for a cooperative framework of mas reorganisation. In *SBIA*, pages 506–515, 2004.
- [76] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.
- [77] IBM. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001.
- [78] C. A. Iglesias, M. Garijo, J. C. Gonzalez, and J. R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. *Intelligent agents IV*, pages 313–326, 1998.
- [79] Sara Jones, Neil A. M. Maiden, Sharon Manning, and John Greenwood. Informing the specification of a large-scale socio-technical system with models of human activity. In *REFSQ'07*, pages 175–189, 2007.
- [80] Haruhiko Kaiya, Daisuke Shinbara, Jinichi Kawano, and Motoshi Saeki. Improving the Detection of Requirements Discordances Among Stakeholders. *Requirements Engineering*, 10(4):289–303, 2005.
- [81] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [82] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending Planning Graphs to an ADL Subset. In *ECP'97*, pages 273–285, 1997.
- [83] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. Organizational patterns for early requirements analysis. In *CAiSE'03*, pages 617–632, 2003.
- [84] Elias Koutsoupias and Christos Papadimitriou. Worst-Case Equilibria. In *STACS'99*, pages 404–413, 1999.
- [85] Jeff Kramer. Distributed software engineering. In *ICSE*, pages 253–263, 1994.
- [86] Kevin Lai, Michal Feldman, Ion Stoica, and John Chuang. Incentives for Cooperation in Peer-to-Peer Networks. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [87] Emmanuel Letier and Axel van Lamsweerde. Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. *SIGSOFT Softw. Eng. Notes*, 29(6):53–62, 2004.
- [88] Magnus Ljungberg and Andrew Lucas. The oasis air-traffic management system. In *PRICAI'92: In Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, 1992.

- [89] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. pages 48–57, 1994.
- [90] LPG Homepage. LPG-td Planner. <http://zeus.ing.unibs.it/lpg/>.
- [91] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.
- [92] Jose Luis Mate and Andres Silva. *Requirements Engineering for Sociotechnical Systems*. Infoscl, London, 2004.
- [93] Mihhail Matskin and Enn Tyugu. Strategies of Structural Synthesis of Programs and Its Extensions. 20:1–25, 2001.
- [94] Alistair Mavin and Neil Maiden. Determining socio-technical systems requirements: Experiences with generating and walking through scenarios. In *RE'03*, pages 213–222, 2003.
- [95] Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. On the Topologies Formed by Selfish Peers. In *PODC'06*, pages 133–142, 2006.
- [96] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, 2003.
- [97] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46. ACM, 2000.
- [98] Object Management Group. Model Driven Architecture (MDA). <http://www.omg.org/docs/ormsc/01-07-01.pdf>, July 2001.
- [99] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, 1994.
- [100] Leon J. Osterweil. A future for software engineering? In *FOSE '07: 2007 Future of Software Engineering*, pages 1–11, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [101] Joachim Peer. Web Service Composition as AI Planning – a Survey. Technical report, University of St. Gallen, 2005.
- [102] Axel Polleres. *Advances in Answer Set Planning*. PhD thesis, Vienna University of Technology, Austria, 2003.
- [103] Colin Potts and Wendy C. Newstetter. Naturalistic inquiry and requirements engineering: Reconciling their theoretical foundations. In *RE'97*, page 118, 1997.
- [104] Steve Roach and Jeffrey Baalen. Automated Procedure Construction for Deductive Synthesis. *Automated Software Engineering*, 12(4):393–414, October 2005.

- [105] Gunter Ropohl. Philosophy of socio-technical systems. In *In Society for Philosophy and Technology* 4(3), 1999.
- [106] Winston Royce. Managing the development of large software systems. In *IEEE WESCOM*, 1970.
- [107] Reto Schmid, Johannes Ryser, Stefan Berner, Martin Glinz, Ralf Reutemann, and Erwin Fahr. A Survey of Simulation Tools for Requirements Engineering. Technical Report 2000.06, Institut fur Informatik, University of Zurich, 2000.
- [108] Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Simple and minimum-cost satisfiability for goal models. In *CAISE'04: In Proceedings International Conference on Advanced Information Systems Engineering*, volume 3084, pages 20–33. Springer, June 2004.
- [109] Security and Dependability Tropos Tool. http://sesa.dit.unitn.it/sistar_tool/.
- [110] Serenity: System Engineering for Security and Dependability. Deliverable A1.D2.1. Security and Privacy Requirements at Organizational Level. <http://www.serenity-forum.org/Work-package-1-2.html>.
- [111] Herbert A. Simon. *Administrative Behavior: a study of decision-making processes in administrative organizations*. ???, 2nd edition, 19??
- [112] Herbert A. Simon. *The Science of the Artificial*. MIT Press, 1969.
- [113] Ian Sommerville. *Software engineering (7th ed.)*. Addison-Wesley, 2004.
- [114] Jussi Stader. Results of the enterprise project. In *Proceedings of 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems*, 1996.
- [115] Alistair G. Sutcliffe and Shailey Minocha. Linking Business Modelling to Socio-technical System Design. In *CAiSE'99*, pages 73–87, 1999.
- [116] Eva Tardos. Network Games. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 2004.
- [117] Walter F. Truszkowski, Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. Autonomous and autonomic systems: A paradigm for future space exploration missions. *IEEE Transactions on Systems, Man and Cybernetics*, 36(3):279–291, 2006.
- [118] R.M. Turner and E.H. Turner. A two-level, protocol-based approach to controlling autonomous oceanographic sampling networks. *IEEE Journal of Oceanic Engineering*, 26(4):654–666, 2001.
- [119] Axel van Lamsweerde. Requirements Engineering in the Year 00: a Research Perspective. In *ICSE'00*, pages 5–19, 2000.

- [120] Guy H. Walker, Neville A. Stanton, Paul M. Salmon, and Daniel P. Jenkins. A review of sociotechnical systems theory: a classic concept for new command and control paradigms. *Theoretical Issues in Ergonomics Science*, 2008.
- [121] Michael Weiss, Daniel Amyot, and Gunter Mussbacher. Formalizing Architectural Patterns with the Goal-oriented Requirement Language. In *VikingPloP'06*, 2006.
- [122] Daniel S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.
- [123] K. Williamson and M. Healy. Industrial applications of software synthesis via category theory. pages 35–43, 1999.
- [124] T. De Wolf and T. Holvoet. Towards a methodology for engineering self-organising emergent systems. *Self-Organization and Autonomic Informatics (I)*, 135(1):18–34, November 2005.
- [125] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.
- [126] Eric Siu-Kwong Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1996.
- [127] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):417–470, 2003.
- [128] Nicola Zannone. *A Requirements Engineering Methodology for Trust, Security, and Privacy*. PhD thesis, University of Trento, 2006.
- [129] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *REFSQ'08*, pages 88–94, 2008.

