PhD Dissertation

International Doctorate School in Information and Communication Technologies

University of Trento

Goal-Oriented Development of Self-Adaptive Systems

Mirko Morandini

Advisor:

Dr. Loris Penserini

Fondazione Bruno Kessler, Trento

Co-Advisor:

Dr. Anna Perini

Fondazione Bruno Kessler, Trento

March 31, 2011

Abstract

Today's software is expected to be able to work autonomously in an unpredictable environment, avoiding failure and achieving satisfactory performance. Self-adaptive systems try to cope with these challenging issues, autonomously adapting their behaviour to a dynamic environment to fulfil the objectives of their stakeholders. This implies that the software needs multiple ways to accomplish its purpose, enough knowledge of its construction, decision criteria for the selection of specific behaviours and the capability to make effective changes at runtime. The engineering of such systems is still challenging research in software engineering methods and techniques, as recently pointed out by the research community.

The objective of this thesis is twofold: First, to capture and detail at design time the specific knowledge and decision criteria needed for a system to guide adaptation at run-time. Second, to create systems which are aware of their high-level requirements, by explicitly representing them as run-time objects, thus enabling it to act according to them and to monitor their satisfaction.

To deliver on this aim, we provide conceptual models and process guidelines to model at design time the knowledge necessary to enable self-adaptation in a dynamic environment, extending the agent-oriented software engineering methodology Tropos. The resulting framework, called Tropos4AS, offers a detailed specification of goal achievement, of the relationships with the environment, of possible failures and recovery activities.

A claim underlying the approach is that the concepts of agent, goal, and goal model, used to capture the system's requirements, should be preserved explicitly along the whole development process, from requirements analysis to the design and run-time, thus reducing the conceptual gaps between the software development phases, and providing a representation of the high-level requirements at run-time. A direct, tool-supported mapping from goal models to an implementation in a Belief-Desire-Intention agent architecture, and an operational semantics for goal model satisfaction at run-time, complement this work. The framework is evaluated through application to research case studies and through an empirical study with subjects, assessing the usability and the comprehensibility of the modelling concepts.

Keywords: Agent-oriented software engineering, development of self-adaptive systems, BDI agents, agent oriented programming.

Acknowledgements

This thesis could not have been completed without the precious support of many people. Above all, I wish to thank Loris Penserini, my advisor. Despite he left Trento shortly after the begin of my journey through the world of research, we were able to stay in contact and to have a fruitful collaboration. Loris, this big target was only possible due to your persistent support and your encouraging words. Thank you!

I would like to say sincere thanks to my co-advisor, Anna Perini. She was my local contact person, was always available for discussing my thoughts, and her comments and contributions have always been of great value for me. Profound thanks also go to Paolo Tonella, head of the SE unit at FBK, who gave me the opportunity to work part-time on his projects, supporting me in the last two years of my PhD studies.

I also thank Prof. John Mylopoulos, Prof. Emanuela Merelli and Prof. Manuel Kolp, for having accepted to be in my thesis committee and to spend their precious time for evaluating my work.

My visit at IRIT, Toulouse was a unique experience, both personally and scientifically. Thanks to everyone who made this possible, in the first place Prof. Marie-Pierre Gleizes. Especially, I would also like to thank Angelo Susi, who is able to give precious advice in any situation.

Last but for sure not least, very special thanks go to my colleagues and first of all friends at FBK: Chaira, Cu, Nauman, Alessandro, Andrea and Surafel in the Software Engineering unit, my colleagues in the IBT project, Mariano, Fitsum and Roberto, as well as all my other friends there: Komminist, Leonardo, Martin, Peter, Sepideh, Shiva and many others. You have always been able to cheer me up and to make our offices a pleasant place to stay!

Schlussendlich möchte ich jenen ganz besonders danken, die mir am meisten am Herzen liegen: meinen Eltern, die mich über all die Jahre hinweg immer unterstützt haben, und denen ich jetzt endlich sagen kann: “Jo, I hon fertig studiert!”, und meiner Freundin Simone, die mir die vielen Jahre hindurch immer wieder Kraft und Mut gegeben hat, mich in traurigen Situationen unterstützt und in fröhlichen Momenten begleitet hat. Wir beide schließen in diesem März unser Studium ab. Ich freue mich, mit dir in einen neuen Abschnitt unseres Lebens eintreten zu können!

Mirko

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's software is expected to be able to work autonomously in an open, dynamic and distributed environment, being available anywhere and at any time.

Self-adaptive software systems were proposed by different research communities as a solution to cope with uncertainty and partial knowledge about the environment, autonomously adapting their behaviour to a changing environment, to fulfil the objectives of their stakeholders. Following [Ganek and Corbi, 2003], we define self-adaptive systems as *systems that can automatically take the correct actions based on their knowledge of what is happening in the systems, guided by the objectives, the stakeholders assigned to them.* In other words, self-adaptive software can modify its behaviour in response to changes in its operating environment to better do what the software is intended for. This implies that the software has multiple ways of accomplishing its purpose, has enough knowledge of its construction and the capability to make effective changes at runtime [Laddaga, 2006].

Concerning the development of such systems, it may become infeasible to accommodate dynamic and unforeseen changes to system requirements and design using traditional software engineering processes, since the engineer may have partial and uncertain knowledge about run-time scenarios. Ideally, the software would have to autonomously adapt to different circumstances,to satisfy the stakeholders' requirements in an optimal way, shifting various decisions which traditionally have been made at design-time, to run-time [Zhu et al., 2008]. Thus, the development of self-adaptive software calls for new design and programming concepts, which have recently been pointed out and discussed in the research community [Cheng et al., 2009a, Di Nitto et al., 2008, Salehie and Tahvildari, 2009, Sawyer et al., 2010].

Several requirements analysis and software programming approaches introduce human-oriented abstractions such as *agents* and *goals* and seem thus valid candidates in order to manage the development of such systems: goal-oriented requirements engineering aims at capturing stakeholder goals, abstracting from structural details, while available agent-oriented programming frameworks provide a basis to build autonomous agents, which are supposed to be able to pursue their goals under unpredictable environmental circumstances. However, in existing agent-oriented engineering methodologies, despite the use of similar concepts in requirements engineering and in implementation – agents with their goals and capabilities – still conceptual gaps exist between the goal-oriented requirements models, the design and the implementation.

1.1 Challenges in Developing Self-Adaptive Systems

In a joined effort, participants coming from various research communities elaborated a preliminary research roadmap, which highlights the differences of self-adaptive software in comparison to traditional software and identifies new challenges to be addressed for engineering self-adaptive systems [Cheng et al., 2009a].

From a modelling perspective, self-adaptive systems would need a decentralised, scalable design, possibly incorporated with AI techniques. From an architectural viewpoint, a self-adaptive system needs to implement some form of built-in feedback loop such as Kephart's MAPE loop [Kephart and Chess, 2003], collecting information, analysing it, deciding on further actions and performing them. This loop should be explicit in the system, separating concerns of the system functionality from concerns of self-adaptation.

Requirements engineering for self-adaptive system would have to cope with incomplete and changing requirements, involving various degrees of variability and uncertainty [Cheng et al., 2009b]. In traditional software engineering, decisions on requirements alternatives are taken in the early phases of development. In an open, changing world, applications cannot be specified completely, and many decisions that traditionally have been made at design-time, have to be shifted to run-time [Di Nitto et al., 2008, Zhu et al., 2008]. A development process tailored towards adaptivity should be able to leave these decisions to the software at run-time, giving it the possibility to adapt to various, possibly unexpected situations.

Thus, a self-adaptive system needs knowledge about the goals to reach, about its capabilities, about how to combine and use them to reach its goals in a specific context, and about how to (re)act if something goes not as expected. This domain knowledge is hard to be grasped or generated by any reasoning or learning method at run-time. Hence, the design model should provide the system with the contextual and social information needed for suggesting the behaviour the system can assume in order to cope with different situations at hands. Namely, the designer needs to give "hints" and to "guide" the system in properly interpreting contextual information in order to decide about *when* to change its behaviour and *which* alternative behaviour to select.

Moreover, to cope with changing circumstances, recent research on self-adaptive systems points out the importance of the traceability of requirements in the implementation, and the need for an explicit representation of requirements as run-time objects. Being aware of its requirements at run-time, a system would be able to monitor them, to reflect upon them and to guide its behaviour according to them [Cheng et al., 2009a, Sawyer et al., 2010]. In this way, the system would be able to manage the variability in the requirements and the alternatives resulting thereof at run-time, like the designer would have done at design time. This also sets up a basis for requirements changes at run-time, where the system itself, implementing learning and user profiling techniques, recognizes the need for a change in its requirements.

1.2 Objectives and Approach

The objective of this thesis is twofold:

First, we want to capture and detail at design time the specific knowledge and decision criteria that guide adaptation at run-time, for a system that has to work autonomously in a dynamic environment.

Second, we aim at bringing the high-level requirements to run-time, to make a systems aware of its requirements, thus enabling it to act according to them and to monitor their satisfaction.

To deliver on our first objective, we adopt an agent-oriented engineering approach to provide a process and modelling language that captures at design time the knowledge necessary for a system to deliberate on its goals in a dynamic environment, and thus enabling a basic feature of self-adaptation. We propose a software model that integrates the goals of the system with the environment, and an iterative development process for the engineering of such systems, that takes into account the modelling of the environment and an explicit modelling of failures.

Taking ideas from organisational modelling [Yu, 1995] and goal-oriented requirements engineering [van Lamsweerde, 2001], we use *goal models* as the main building block throughout the whole development process, from requirements engineering to design, implementation and run-time. Goal models represent the *rationale* of the system and define possible goal decompositions and the relation of the system's functionalities with its high-level goals.

For our second aim, a basic claim in our work is that the concepts of *agent*, *goal*, and *goal model*, used to capture the system's requirements, should be preserved explicitly along the whole development process, from requirements analysis to the design and run-time, to lower the conceptual gaps between the software development phases, and to provide an explicit representation of the high-level requirements at run-time. Being able to represent functional and non-functional requirements, alternative choices, conflicts and responsibilities, *Goal Models* are claimed to be a valid candidate also for representing requirements at run-time [Sawyer et al., 2010].

By an explicit representation of goals in all development phases, a complete translation of requirements concepts to traditional software level notions, such as classes and methods of object-oriented programming, is avoided. This choice contributes to a smoother transition between the development phases, reducing loss and conceptual mismatch and simplifying the tracing of decisions made in requirements analysis and design to the implementation.

For the implementation we rely on agent systems with a BDI (Belief-Desire-Intention) architecture, which have a native support for the concepts of agent, goal, plan and belief and include basic monitoring and adaptation abilities. Defining a direct mapping from requirements artefacts (the goal models) to implementation concepts, and a supporting middleware, we create an explicit, navigable and monitorable representation of the requirements at run-time. This explicit representation of the requirements at run-time can however be realised effectively only by tools supporting the mapping and code generation, it needs a platform for goal models at run-time and moreover calls for the definition of run-time semantics for their satisfaction.

In pursuing our objectives, we focus on (and limit to) the design of the information needed for a system's decision-making process, the so-called *knowledge level*, leaving out the detailed modelling and implementation of the single functionalities (*capabilities*) of the system. Moreover, we aim at providing a run-time framework which gives to the software awareness about its requirements. The optimisation of a system's behaviour, by the use of run-time goal model reasoning, learning and knowledge acquisition strategies, is not part of, but would be complementary to our work.

1.3 Contributions

In this thesis we define *Tropos4AS*, a framework which provides a modelling language and a process tailored to the development of self-adaptive systems.

We adopt the agent-oriented software engineering methodology *Tropos* [Bresciani et al., 2004a] for requirements modelling and extend *Tropos* goal models with a model of the system environment and with an explicit description of an agent's goal achievement in correlation to the environment (conditions modelling). Additionally it is possible to model details in the goal achievement process, for goal sequence and conflict. Moreover, we define a set of models and a process to analyse possible failures in a system, errors causing them and associated prevention or recovery activities. The aim of this is to elicit missing requirements and capabilities needed in a changing environment, for the purpose of increasing the robustness and fault tolerance needed in a changing environment.

We aim at lowering the conceptual gaps between requirements, design and implementation, avoiding a translation of requirements concepts to traditional software level notions such as classes and methods of object-oriented programming. Our approach uses the same 'mental' concepts of *agent* and *goal* in all phases from requirements analysis to the design and run-time, and defines a design process and a direct (automated) mapping from the design to implementation concepts.

Specifically, we adopt the BDI agent language and run-time platform *Jadex* [Pokahr et al., 2005], which provides an explicit representation of goals at run-time. A tool for modelling our extended goal models and for mapping them directly to agent code endowing the same structure, together with a supporting middleware, are provided. We also define operational semantics for the satisfaction of goals in goal models, which explain the intended behaviour of goal achievement at run-time, and are customisable for a formal definition of the behaviour of platforms supporting goal models at run-time.

Self-adaptation can in certain domains also be achieved by an emergent behaviour obtained by *self-organisation* of the components in a system. We sketch an approach for modelling self-adaptation by self-organisation, which combines models and process of the goal-oriented "top-down" methodology *Tropos4AS* with the "bottom-up" methodology for cooperative agent communities ADELFE [Bernon et al., 2005].

An evaluations of the effectiveness of the *Tropos4AS* framework is provided along different lines: through the application to examples of the process and the mapping; by a simulation of generated agent prototypes, for performing a preliminary investigation about an improvement of goal models through run-time feedback; and through an empirical study with subjects, which consists of two controlled experiments evaluating the effectiveness and the comprehension of the modelling language in comparison to *Tropos*, while performing modelling and model understanding tasks.

Parts of the contents of this thesis were published in international conferences and workshops. Goal-oriented modelling of self-adaptive systems, with a first version of the *Tropos4AS* framework, is presented in [Morandini et al., 2008d] and [Morandini et al., 2008c] and detailed in a technical report ([Morandini et al., 2010]), while the tool-supported mapping to an implementation, based on a prototype presented in [Morandini, 2006], is treated in [Penserini et al., 2007a], shown for *Tropos* on an example in [Morandini et al., 2008a], and extended for *Tropos4AS* in [Morandini et al., 2008b]. The operational semantics for goal model satisfaction at run-time are presented in [Morandini et al., 2009b], while [Morandini et al., 2009a] treats adaptivity by self-organisation and [Morandini et al., 2008e] gives an application of the mapping and shows how feedback from run-time can contribute to an improvement of the design.

1.4 Outline of the Thesis

Chapter 2 presents the state of the art in the research areas addressed by this thesis, namely adaptive systems, goal- and agent-oriented software engineering, and concepts in programming of agent systems. Moreover, necessary background for the comprehension of this thesis is recalled.

Chapter 3 introduces a new modelling framework, extending *Tropos* goal models by concepts useful for capturing the interplay of the system with its environment and for detailing goal achievement and alternatives selection dynamics, basic information necessary to capture the main characteristics of systems that call for a self-adaptive solution. Modelling steps are then defined, to elicit necessary information, to analyse and to detail the newly introduced abstractions.

In Chapter 4, these modelling steps are integrated into a tool-supported process, which spans the development phases until the implementation, focussing on knowledge level artefacts. A mapping from design artefacts to a BDI-based implementation is detailed, which preserves the goal models at run-time, and tool support for modelling and automated code generation for the *Jadex* agent platform, are presented.

In Chapter 5, an operational semantics for a subset of the introduced goal model extensions is defined, to formalize the interpretation of extended goal models and to give the basis for a formal definition of the behaviour of agent platforms supporting goal models at run-time.

Chapter 6 opens a further research direction, presenting a first attempt to adapt the framework for the development of self-organising multi-agent systems. It proposes the combination of the introduced, *Tropos*-based "top-down" engineering approach with the "bottom-up" approach ADELFE, resulting in multi-agent systems which act adaptively by self-organisation.

The chapters 7 and 8 report experiments for assessing the presented framework. In Chapter 7, the feasibility of the development process is shown on examples, and the possibility of bringing feedback collected during the usage of a system created following the presented process, is evaluated. Chapter 8 shows the results of an empirical study with subjects, which evaluates the usability of the newly introduced modelling concepts through two experiments.

Chapter 9 summarises the contributions of the thesis, draws a conclusion, identifies open points and highlights future work directions.

Chapter 2

State of the Art & Background

This chapter reviews relevant literature from the main areas our research builds on and introduces background work necessary for understanding this thesis.

Research in self-adaptive systems is carried out in various domains, from AI and control theory, to adaptation algorithms and architectures. Moreover, it is widely recognised that the development of self-adaptive systems brings challenges to virtually every phase of software engineering, asking for a paradigm shift in requirements elicitation and analysis, in the system's architecture and design, in implementation and at run-time.

In the context of our work we limit our review along the following three lines (Sections 2.1-2.3): main approaches for the analysis and design of self-adaptive systems, with emphasis on goal-oriented requirements engineering, variability design, anticipation of failures, and self-adaptivity at run-time; software agents as a promising implementation platform for self-adaptive systems; and agent-oriented software engineering methodologies as a starting point for the development of self-adaptive systems. Section 2.4 gives an introduction and explains the important aspects of three works which are used as a baseline for this thesis: the agent-oriented development methodologies *Tropos* and ADELFE and the agent-oriented implementation framework Jadex.

2.1 Analysis and Design for Self-Adaptive Systems

Adaptivity has to be seen as characteristic of a solution to the problem of satisfying stakeholders' needs, taking into account uncertainty and incomplete knowledge of the environment, non-determinism in the environment, incomplete control of components (also due to humans in the operating loop), or requirements best satisfiable by solutions emerging from the interplay in decentralised systems [Müller et al., 2008].

To cope with this problem, first the stakeholders' objectives, but also variability and uncertainty in the problem space (domain and requirements) have to be captured. However, applications cannot be specified completely, and many decisions can only be taken during run-time [Di Nitto et al., 2008]. Thus, a self-adaptive system has to shift decisions on variability that traditionally have been made at design-time, to run-time, to meet the needs of the stakeholders, adapting its behaviour to the context (cf. [Zhu et al., 2008]).

This points out the need for a runtime representation of requirements, which is suitable for inspection and possibly also for modification. Thus, it is crucial to make requirements available as runtime objects, causally connecting the requirements models to the executing system, for creating a system aware of its requirements and able to reflect upon itself [Cheng et al., 2009a, Sawyer et al., 2010].

It is widely recognised that the development of self-adaptive systems brings challenges to virtually every phase of software engineering, asking for a paradigm shift in requirements elicitation and analysis, in the system's architecture and design, in implementation and at run-time [Cheng et al., 2009a].

From a modelling perspective, self-adaptive systems would need a decentralised, scalable design, possibly incorporated with AI techniques. From an engineering viewpoint, to enact the adaptation, a self-adaptive system needs to implement some form of built-in feedback loop such as Kephart's MAPE[1] loop, collecting information, analysing it, deciding on further actions and performing an adaptation if necessary. This loop should be explicit in the system, separating concerns of the system functionality from concerns of self-adaptation (cf. [Sawyer et al., 2010]). Several engineering questions arise for each stage of this loop, in particular for decision making, on how to decide among various alternatives, to adapt to reach a desirable state.

[Kephart and Chess, 2003] identify various challenges in engineering self-adaptive systems, in the life cycle of individual autonomic elements, the relationship among elements, and the whole context of the system, up to the interface between humans and the systems. They also emphasize that representing needs and preferences will be just as important as representing the system's capabilities. Furthermore, ensuring reason-

[1]In their seminal work on *autonomic computing*, Kephart and Chess [Kephart and Chess, 2003] envision systems composed by *autonomic elements* characterized by a **MAPE** loop, continuously **M**onitoring the managed element and its external environment, **A**nalysing this information, basing on its own actual knowledge of the environment and of past, **P**lanning the following activities and **E**xecuting them.

able system behaviour in the face of erroneous input will be a challenge. Moreover, programmers will need to be supported by tools that help to acquire and represent high-level specifications of goals and map them onto lower-level actions, and tools to build elements that can establish, monitor, and enforce contracts or agreements.

Berry [Berry et al., 2005] categorised the different perspectives of RE which has to be carried out in developing and running a self-adaptive systems. He identifies four levels of requirements engineering for adaptation, where each level corresponds to the objectives of a different stakeholder: **Level 1** comprises traditional requirements engineering, fulfilled by the system developer in order to elicit customers' and users' objectives; **Level 2** considers RE done by the system itself at run-time, to determine if and how to adapt; **Level 3** includes RE that has to be done to determine the system's adaptation architecture, that is, to determine how the system will carry out Level 2 RE. **Level 4** spans research on adaptation in general, e.g. for optimising adaptation mechanisms. Regarding to this classification, to meet the objectives of our work we span the four levels, doing research on software engineering for adaptation, and providing a framework that captures, at Level 1, the specific requirements that characterize the problems which call for adaptivity. Furthermore, at Level 3, a specific architecture will be chosen to enable basic Level 2 adaptation mechanisms at run-time, over the variability captured in Level 1.

To capture the objectives of the various stakeholders and the variability in the problem space, promising requirements engineering approaches for self-adaptive systems base on *goal-oriented requirements engineering (GORE)* notions and organisational modelling, using the human-oriented notion of actor, with its goals and dependencies, for the description of software systems that have to work in a distributed, uncertain environment.

Goal-Oriented Requirements Engineering

In GORE [Dardenne et al., 1993], requirements are elicited, specified and elaborated using the concept of ***goal*** as main abstraction. Goals are objectives the system under consideration should achieve, representing high-level (strategic) concerns.

To go towards a detailed definition of the *system-to-be*, the requirements engineer refines these high-level goals, to address more and more concrete concerns, decomposing goals, finding alternatives tailored to a specific context, and finally discovering means for their achievement. This kind of analysis is supported by *Goal Models*, built by hierarchical AND/OR decomposition of high-level goals, to show the *rationale* of a system.

Various software engineering approaches adopt goal models for requirements modelling, mostly basing their graphical and formal languages for goal models on Lamsweerde's *KAOS* [Dardenne et al., 1993, Darimont et al., 1997] or Yu's *i** modelling framework [Yu, 1995].

KAOS The *KAOS* language was developed specifically for software requirements modelling and gives details on responsibilities and operationalisation of requirements. A goal model is composed of goals arranged in AND/OR graphs where a goal node can have several parent nodes as it can occur in several decompositions, called *reductions*. In the *KAOS* metamodel, the *reduction* meta-relationship allows for goal refinement and for modelling alternative ways of achieving goals. Goals are defined informally (abstract) and refined into more formal (concrete) ones until reaching subgoals (leaf-goals) that can be operationalised through *constraints*, formulated in terms of objects and actions. The semantics of the goal model are described by a first order temporal logic that allows for a formal verification of requirements.

***i** modelling framework** The *i** modelling framework [Yu, 1995] uses GORE concepts to model the organisational environment, dependencies and responsibilities in a system. Using the concepts of *actor*, *goal* and *dependency*, the rationale (the *why*) behind the requirements of a system-to-be are captured, including not only functional, but also non-functional and quality-of-service related requirements, modelled by *soft-goals*. *Tasks* are defined as a means to achieve goals, while task decomposition links provide details on the tasks and the (hierarchically decomposed) sub-tasks to be performed by each actor. Alternative tasks can be modelled as means for achieving a goal. *i** focuses on the early development phases of eliciting the stakeholders, understanding their needs and responsibilities in the organisation and deriving the goals of the system-to-be, although it was still developed without agent-oriented programming in mind [Bresciani et al., 2004a]. Various successive approaches use *i** for a goal-oriented analysis of functional and non-functional requirements, to complement object-oriented analysis, and also specifically for capturing the requirements of self-adaptive systems.

Reasoning on such goal models to find the tasks that maximize goal (and soft-goal) satisfaction, or to test satisfyability of (a group of) goals, performed off-line, is addressed in various works, e.g. [Giorgini et al., 2004] or [Fuxman et al., 2001], and recently, specific to variability in early requirements analysis, in [Jureta et al., 2010]. The work in this thesis does not go into this direction – the implemented simple reasoning algorithms serve for illustration and prototype execution.

Dealing with variability and uncertainty in the requirements

The general objective of goal modelling is to refine goals so that the set of subgoals that satisfy their parent goal is necessary and sufficient. When uncertainty, both in the requirements and in the domain, exists, subgoals will never be sufficient [Cheng et al., 2009b]. Uncertainty must be handled, therefore, by capturing alternatives and defining the opportunity for selecting between them, and by assigning responsibility to a human agent or by introducing some adaptive behaviour into the software.

In the context of requirements analysis and design for self-adaptive systems, alternatives in the problem space (domain and requirements) and in the solution space (system behaviours) have to be made explicit. Methods for modelling alternatives are provided by GORE and in particular by the following research.

The work by Lapouchnian et al. [Lapouchnian et al., 2006] can be categorised at Berry's level 1. It enriches $i*$ models to obtain a design-level view, aiming at a specification of autonomic systems. Annotations such as sequence, priority, and conditions are introduced for decompositions and expressions can define variable contribution to softgoals. The models obtained are a first step towards using goal models for a more detailed design of the system. Nevertheless, the environment and its influence on the system behaviour can be only modelled at high-level through $i*$ delegation, goals lack of important run-time concepts such as a life cycle with creation and achievement, and no development process is defined.

Yijun Yu et al. [Yu et al., 2008] capture high variability in the requirements using Lapouchnian's annotated goal models. At design time, they are mapped to feature models, distinguishing run-time variability (dependent on some input) from design-time variability, where selection can be made at design time, based on quality criteria (softgoals).

Liaskos et al. [Liaskos et al., 2006] propose a formal language to specify stakeholder preferences and to reason about them with the purpose of supporting the analysis of behaviour variability at the requirements level. Goal models have been found to be effective to study stakeholders' goals and have been annotated to capture background variability, i.e. changes in the environmental context which would affect goal selection and achievement, in the problem space. Variability which can not be resolved at design time is modelled in the solution space with feature-models, following product-line engineering approaches. It has been left as future work to show how goal model variability at requirements level influences the run-time behaviour of a software system. With the aim of modelling the requirements of mobile systems in a changing environment, variability in the context is also captured by Ali et al. [Ali et al., 2010], completing goal models with location-based annotations.

Other approaches augment goal models with details of the goal-satisfaction behaviour of an agent situated in a dynamic environment. Representatively, we report the *Goal Decomposition Tree* (GDT) model [Simon et al., 2005], recently extended to multi-agent systems [Mermet and Simon, 2009], in which each goal is specified by a type (state and progress goals), a satisfaction condition and a guaranteed property in case of goal failure. Moreover, GTD models include relationships to express subgoal iteration and sequence; besides, decomposed goals can be defined as ”lazy” or needing ”necessary satisfaction”. A complete formal specifications of a single software system is obtained (with similarities to *Formal Tropos* [Fuxman et al., 2004], presented in more detail later in this chapter), which can be translated into an automata which implements the behaviour specified by the GDT. However, the effort of fully specifying goals and actions, with their pre- and post-conditions, is not feasible for bigger systems in an undefined environment.

Besides capturing variability and selection criteria, also dealing with uncertainty in requirements models becomes important [Sawyer et al., 2010], because it cannot be assumed that requirements for all possible environments can be anticipated, and thus all adaptations will be known in advance. As a result, requirements for self-adaptive systems may involve degrees of uncertainty and incompleteness. This could be achieved by “relaxing” the typical prescriptive notion of “shall” in requirements definition. *RELAX* [Whittle et al., 2009] analyses which requirements can be ’relaxed’ (i.e. they have not to be satisfied under any condition) to gain some degree of freedom to be used for adaptivity. This “relaxation” is textually modelled by using a requirements vocabulary able to express uncertainty. With the modal verb *MAY*, possible alternatives can be specified, while temporal uncertainty can be expressed by operators such as *AS EARLY AS POSSIBLE*, *EVENTUALLY*, *UNTIL*, and ordinal uncertainty e.g. by *AS MANY*, and *AS FEW AS POSSIBLE.* However, with the high expressiveness and freedom given by RELAX, it becomes difficult to define concrete goals and precise contracts out of requirements defined in such way.

Modelling of self-adaptive systems

More recent works attempt to address specifically the nature of requirements for self-adaptive systems, which requires to understand what are the specific knowledge and the decision criteria that guide adaptation at run-time – namely the issues that define the first research objective in this thesis work – and how they can be made explicit. In part, these works propose entire development processes, which include adaptation at run-time.

Requirements monitoring [Feather et al., 1998] aims at verification of the run-time behaviour of a system, tracking its behaviour for deviations from its goal-oriented KAOS requirements specification. The system alerts the user or tries pre-defined run-time reconfiguration tactics, if it recognizes that it has to work in an environment for which it is not designed. To do this, KAOS models have to be formalized and the combinations of events to monitor have to be defined using temporal logic.

For [Goldsby et al., 2008], a self-adaptive system is assumed to be a collection of steady-state systems, one of which is executing at a given point in time. The system uses *i** for modelling every possible system configuration in a distinct goal model, together with characterising the conditions for transition between two system configurations, by a goal model for every potential transition. The main difference with respect to our objective is that we try to avoid to specify every system and every possible adaptation between systems, which may shortly lead to a bottleneck and to numerous, mostly redundant models. Instead, we model a single system that includes all variability, at high and low levels. Similarly, also Berry [Berry et al., 2005] proposes goal-oriented modelling of every possible system configuration in a distinct goal model (using KAOS), whereas our approach captures the variability needed for adaptation, in a single model.

Cheng and Bencomo [Cheng et al., 2009b] recently presented a goal-based modelling approach to develop the requirements for a self-adaptive system, giving the focus on explicitly factoring environmental uncertainty into the process and resulting requirements. In an iterative process, first, top-level goals are decomposed and a conceptual domain model identifying the key physical elements of the system and their relationships (e.g., sensors, user interfaces) is built. Basing on KAOS goal models and *obstacle analysis* [van Lamsweerde and Letier, 2000], environmental conditions that pose uncertainty at development time are identified, to uncover requirements that need to be updated. Possible failures are mitigated by adding low-level subgoals, using their RELAX language [Whittle et al., 2009] to make existing requirements more flexible, and by adding new high-level goals which define a new target system to which to adapt.

In their recent work, Baresi and Pasquale [Baresi and Pasquale, 2010] follow an approach similar to ours [Morandini et al., 2008d], adopting the KAOS methodology. Taking inspiration from KAOS *obstacle analysis*, the approach models *adaptive goals*, which identify conditions for goal violation and define various recovery strategies, acting on the goal model at run time. Ongoing work aims at instantiating the framework for a service platform.

Qureshi et al. [Qureshi and Perini, 2009], in their *"adaptive requirements"* framework, capture requirements in goal models and link them to an ontological representation of the environmental context, to capture alternatives, together with their monitoring specifications and evaluation criteria, aiming at service-based applications.

Conceived for an early requirements analysis, *Techne* [Jureta et al., 2010] combines goals, quality constraints and preferences in a single model. Candidate solutions to the requirements problem (containing tasks and quality constraints) are derived, and sorted by number of preferences satisfied and number of options included. Qureshi et al. [Qureshi et al., 2010a] adopt Techne in their continuous adaptive RE framework, to reflect user-requested run-time requirements changes directly to an explicitly represented requirements model and to adapt to new candidate solutions.

Mylopoulos proposes *awareness requirements*[2], meta-specifications of other requirements in a goal model, which capture the uncertainty about the success and failure of other goals, e.g. by defining the required success rate.

Anticipation of failure

In order for a self-adaptive system to behave as required, it needs to avoid failure, often caused by the unpredictable nature of the environment. Several approaches propose formal analysis techniques with a complete coverage of the scenarios derived from a model of the system, to verify the completeness of the system's requirements. To anticipate failure, [van Lamsweerde and Letier, 2000] deals with the analysis of *obstacles*, sets of undesirable behaviours, which prevent goal achievement. Obstacles are identified and resolved at design-time, exploiting a formalization of goal models in temporal logic, an approach which may become infeasible for large, complex systems. In this work, also a risk analysis is performed, based on the likelihood of obstacle occurrence and on the severity of its consequences. Such an analysis goes beyond the aim of our

[2] John Mylopoulos, *"Awareness Requirements"*. Invited talk at SEAMS'10. See [Souza et al., 2010] for further details.

work, which focuses on design aspects. Risk analysis might however be complementary to our approach of failure modelling, as a means to decide if likelihood and severity of a failure demand a design and implementation as part of the agent's nominal behaviour, or as part of exceptional failure recovery, or that it has not to be implemented at all, because of its low risk.

Formal Tropos [Fuxman et al., 2001] defines a first order temporal logic specification of a *Tropos* requirements model, which allows to perform verification by analysing huge number of scenarios with the purpose to find those which may lead to system failure (i.e. counterexamples). Instead, our main concern is on how to model requirements for adaptive systems which may foresee and manage potential failures at run-time.

The concept of *antigoal* [van Lamsweerde, 2004] is an *obstacle* obtained by negating security-related goals and is thus related to some attacker agents, which might benefit from it. This approach differs from and complements our work at the same time. In fact, despite we do not consider goals belonging to malicious agents, we deal with agent failures from the developer viewpoint.

A tool-supported approach to automated monitoring and diagnosis of requirements is presented in [Wang et al., 2007]. Goal models, goal pre- and postconditions (effects) and logging outputs (obtaining by instrumenting the program) are mapped to propositional SAT formulae. The satisfaction of software requirements is monitored at run-time. For errors, a valid diagnosis can be found by verifying satisfiability of the monitoring output in union with possible diagnoses.

Kiessel et al. [Kiessel et al., 2002] address failure modelling, arguing that it is not feasible to anticipate and prevent all possible errors in such systems. The approach attempts to give a software system the capability to recognize incorrect behaviour and to initiate recovery actions, to avoid failure. The system monitors itself to detect symptoms related to errors, and can execute recovery actions. In alternative, to avoid to get stuck, the system tries different available actions in the domain of possible failures, and learns from their success. However, the modelling of systems which endow such features is not addressed.

Architectures for adaptivity at run-time

Various recent approaches deal with challenges resulting from the implementation of a self-adaptive system. They try to resemble the structure of goal models or to represent important parts of the requirements at run-time, to guide execution and to monitor goal satisfaction – namely, issues that define the second research objective in this thesis.

For example, Nakagawa et al. [Nakagawa et al., 2008, Nakagawa et al., 2010] present a goal-oriented development process which makes use of a requirements model (based on the KAOS methodology) to construct an architectural model for self-adaptive systems. This work presents a mapping of each goal component to a *behaviour* of a JADE agent [Bellifemine et al., 2007], which is implemented to resemble the goal achievement behaviour and the goal hierarchy. However, in this way an explicit representation of the goal model is lost and a goal-directed behaviour, as demanded in this thesis, becomes impracticable.

The goal-driven approach for self-adaptive systems proposed by Salehie [Salehie, 2009] does not origin in requirements engineering, but represents a run-time goal model specific to reason upon for decision-making, i.e. action selection. The work points out that goal models can, as an important difference to rule-based decision making, be navigated at run-time, and policies and priorities can be defined and tuned for achieving the desired behaviour. Furthermore, goal models allow for various, possibly multi-objective, decision-making mechanisms to work with. Zhu et al. [Zhu et al., 2008] analyse goal models for the implementation of autonomic elements in Kephart and Chess' vision of autonomic computing, and propose to realize subtrees of a goal models by autonomic elements.

A middleware-based approach, RAINBOW [Cheng et al., 2004] presents a framework that supports self-adaptation by monitoring of a legacy system with probes and gauges. At run-time, if system constraint violations are detected, adaptation strategies, coded into the middleware, are applied. A a new configuration can then be enforced, which better suites run-time needs. A work with a similar approach has been recently proposed in [Dalpiaz et al., 2009]. Dalpiaz et al. present an architecture for self-reconfiguration of agents which can adapt to different variants. They are autonomous in their decisions, but have social *commitments* to a central entity. Such commitments emphasize the social aspect in agent organisations.

[Qureshi and Perini, 2010] aims at defining a framework for self-adaptive systems, in which the system plays the role of the analyst, and requirements can be requested by users at run-time, through service requests, called "run-time requirements artifacts". The envisioned system is able to look up for available services to operationalise the new requirements, instantiating a MAPE loop for service acquisition and selection.

2.2 Software Agents

Software that realizes goal-oriented requirements is expected to work in an autonomous, goal-directed way. It would have "choice of behaviour", thus it can be seen as an agent [van Lamsweerde, 2001].

The Agent Oriented paradigm (AO), introduced by Shoham [Shoham, 1993], aims at dealing with the increasing complexity of applications, which have to operate within unpredictable, evolving and heterogeneous environments, being capable of flexible, autonomous and proactive action in order to meet its design objectives. It offers thus an appropriate paradigm for the description of engineering and implementation of the self-adaptive systems we consider in our work.

As defined in [Wooldridge, 1997], an agent is defined as an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives. More precisely, autonomous agents are systems that inhabit a dynamic, unpredictable environment in which they try to satisfy a set of time-dependent goals or motivations [Maes, 1994]. If unexpected events occur, Agents are supposed to have the autonomy and proactiveness to try alternatives [Jennings, 2000].

2.2.1 Multi-Agent Systems

Multi-agent systems (MAS) are systems composed of several autonomous agents that can work collectively to reach individual or common goals that are difficult to achieve by an individual agent or a monolithic system, following the idea of a human organisation. Complex or distributed systems can be modelled as MAS to achieve a better decomposition of a problem, or as a natural way to model societies of interacting entities. A MAS could also include human actors, services and legacy systems, which are not necessarily conceived as agent-oriented applications.

Each single agent in a multi-agent system is autonomous and without centralized control and can interact with others through message passing or through activity in their environment, which are both supported, in different flavours, by most available agent languages. Multiple agents could depend one from another for a specific activity, they could delegate goals and activities to others, and share their environmental knowledge. MAS can exhibit an *emergent* behaviour as a result of the cooperation of the single agents.

2.2.2 Agent-Oriented Programming

In Agent-Oriented Programming (AOP) we talk of mental states and beliefs instead of machine states, of plans and actions instead of procedures and methods, of communication and social ability instead of I/O functionalities [Bresciani et al., 2004a].

Agent frameworks and run-time platforms try to abstract from the object-oriented paradigm, introducing agent-oriented concepts and extensions, for creating software which is supposed to have social abilities, be autonomous and proactive. As a main distinction to object-orientation, as lowest common denominator they provide an independent thread of control, built-in messaging functionalities and rely on a middleware that delivers basic services for the realization of multi-agent systems, such as communication protocols and -services. Such basic properties are provided for example by the *JADE* agent platform [Bellifemine et al., 2007], a mature project widely used as a basis for agent systems and platforms.

BDI Agents The predominant approach to the implementation of 'rational' agents, acting goal-oriented, proactive and autonomously, is the BDI (Belief, Desire, Intention) model. It roots in the theory of practical reasoning by Bratman [Bratman, 1987] and provides an architectural basis for agent systems. For an executable model, Rao and Georgeff introduce the more concrete notions of goal, plan and belief for the definition of an agent's behaviour [Rao and Georgeff, 1995].

Goals are defined as desires an agent tries to fulfil (in several definitions they need to be consistent with each other), while plans contain the means of achieving certain future world states (i.e. the possible actions to execute in order to reach a goal). These notions are adopted by several agent-oriented implementation languages and frameworks, such as PRS, AgentSpeak, 3APL, Jack, Jadex, and Jason [Bordini et al., 2005]. Some of these frameworks have an explicit notion of goal for agents at run-time, others rely on the concept of *event* as the driving force of agent activity.

BDI agents endow a reasoning cycle, monitoring the environment to update their belief, deliberating about the goals to achieve, selecting suitable plans and executing them to achieve the goals.

Formal agent definition languages, such as AgentSpeak, 3APL, and GOAL provide formal operational semantics for a BDI agent architecture. As an example, in AgentSpeak, agents are defined by a set of first-order logic formulae, implementing the initial state of the agent's *belief base*, its *goals*, and a library of *plans* that can be executed in reaction to an *event*. The execution platform *Jason* provides an interpreter for AgentSpeak and enables practical development of such agent systems [Bordini and Hübner, 2005]. However, in Jason, goals do only trigger events and are not persistent in time.

The 3APL language [Hindriks et al., 1999] is an agent specification language which defines beliefs, goal base, reasoning rules, and a set of basic actions, where beliefs are a set of *Prolog* rules and facts and actions can be implemented in Java. It offers precise operational semantics, which allow for logical specification and verification. The similar GOAL language [Hindriks et al., 2000] includes persistent, declarative goals instead of goal events.

The frameworks Jack and Jadex provide imperative, Java-based languages for implementing and running BDI agent systems. Jack agents [Howden et al., 2001] are software agents with plans and events, which trigger plans, in an extended and annotated Java-based language. The *Jadex BDI agent framework* [Pokahr et al., 2005] allows agent developers to implement agents, which exhibit a rational, goal-oriented behaviour. Jadex addresses the limitations of other BDI systems by introducing new concepts such as an explicit representation of goals and a goal deliberation mechanisms. Moreover, agents are implemented with well established technologies such as XML and Java. Details can be found in Section 2.4.2.

Semantics of agent languages The semantics of available agent programming languages are either defined formally or determined by their implementation in a conventional language. A core part in agent languages, especially in languages such as GOAL, Jack and Jadex, which include declarative goals, is the the definition of various types of goals. These types define the behaviour that an agent exhibits for trying to achieve these goals Dastani [Dastani et al., 2006] made an effort for categorizing the goal types available in the various languages, while Riemsdijk [van Riemsdijk et al., 2008] and Thangarajah [Thangarajah et al., 2010] give an unifying operational semantics in order to give a solid definition for these goal types. The operational semantics for goals which are collocated in goal models, presented in Chapter 5, are built on top of Riemsdijk's work, which considers only "leaf goals", that is, goals that were directly operationalised by plans.

2.3 Agent-Oriented Software Engineering

The new abstractions introduced with agent-oriented systems cannot be properly captured using a traditional object-oriented design approach, which model the system from an architectural point of view. In particular, an agent's flexible, autonomous goal-oriented behaviour and the complexity of the dependencies and interactions cannot be adequately captured by such approaches. Agent systems should be conceived from a much more natural and social point of view, modelling the actors and objectives of a system explicitly in the requirements elicitation phase.

To capture these new abstractions, several **agent-oriented software engineering (AOSE)** methodologies were proposed in the last years, trying to assist the designer in building software with autonomous, goal-oriented behaviour, starting from requirements analysis and gradually refining and complementing the high-level concepts initially modelled, until reaching software detailed design and implementation.

AOSE Methodologies such as MaSE, Prometheus, and *Tropos* [Henderson-Sellers and Giorgini, 2005] start from a GORE approach in order to figure out stakeholders' intentions (requirements). These are then specified and detailed, evolving to architectural and detailed design and eventually to an implementation of agent systems. The different methodologies adopt different high level design concepts in order to abstract from complex system requirements, and support different design phases.

In the following we briefly mention some AOSE methodologies, which are interesting as background and for comparison to our work, using goal models in some development phase, including a representation of the environment or an elaborated implementation phase. The methodologies *Tropos* and ADELFE are important as baseline for our approach and thus defined in more detail, in Sections 2.4.1 and 2.4.3, respectively. Later in this section we also specifically focus on an approach for the representation of the environment in agent methodologies.

The Multi-agent Systems Engineering methodology (*MaSE*) [DeLoach et al., 2001], extended to *O-MaSE* for organisational modelling [García-Ojeda et al., 2007], was designed to develop general purpose, closed, heterogeneous multi-agent systems. Requirements analysis leads to a Goal Model for Dynamic Systems (GMoDS) [DeLoach and Miller, 2009], with AND/OR refinements, precedence relationships and triggering events. The identified goals are then translated to scenarios described textually and by UML-like diagrams. However, with this translation, the concept of goal and the structure captured in the goal model are lost, also eventually going to an implementation. Hence, it becomes difficult to motivate implementation choices and run-time choices by earlier phases, and to trace modelling decisions from the analysis phase to design and implementation.

O-MaSE goal models are implemented by assigning the leaf goals to agent roles in the MAS [Oyenan and DeLoach, 2010]. By this process, high level information of goal decomposition is no more present at run-time and furthermore the goal model cannot be used as means to detail the steps necessary for the achievement of a single role's (or agent's) goal. The provided MAS simulation environment moreover does not make use of goal-oriented technology.

Prometheus [Padgham and Winikoff, 2002] is a methodology developed following software engineering experiences on the commercial agent platform JACK, and makes use of goal models to describe system requirements. After building a goal model in the requirements analysis phase, the designer identifies those goals that are related to system functionalities and delegates them to specific system actors. The architecture of the system is specified grouping needed functionalities (actions) for goal achievement to agents and defining the interactions between them. In the detailed design phase, the agent internals with its actions, events, plans and data structures necessary to achieve its goals, are defined, mostly through a textual description of scenarios. Already at this stage, goals remain merely a high level motivation for the design decisions made, and *events* become the guiding force for plan execution. From the scenarios, UML2 interaction diagrams, protocols and process diagrams are derived, altering the design focus from *goals* to messages and data. In this way, the context of how and when goals are achieved, is lost, and it becomes difficult to attribute agent activities to goals. Thus, at run-time, agents' awareness about their goal model is limited and the designed agent behaviour becomes mainly reactive rather than proactive and deliberative: the agent cannot reason on its goals in order to deal with failures and to choose alternative behaviours.

A refinement of Prometheus [Khallouf and Winikoff, 2009] aims at overcoming these weak points, maintaining the presence of goals throughout the design phase artefacts, e.g. extending interaction diagrams to include them. The agent developer will thus be aware of the goals that directly demand an agent's functionality. However, the approach does not bring to the design the high-level goal model with its variability and thus an agent will not be able to adapt its behaviour by taking decisions on requirements-level alternatives.

Some of the limitations of previously explained methodologies were considered, but never further investigated, in a preliminary work by Kinny et al. [Kinny et al., 1996], which proposed a methodology for the development of BDI agents. Kinny et al. define to different viewpoints for an agent: while the ideas for the so-called external viewpoint (decomposition of the system into agents with responsibilities, available services, and interactions) have been adopted in several prominent AOSE methodologies, first of all, GAIA [Zambonelli et al., 2003], the *internal viewpoint* of an agent, seems not to have been further investigated. The authors proposed several models to define the internals of an agent: a belief model to describe the environment and the internal state, a goal model to capture the goals that an agent may possibly adopt and the events to which it can respond, and a plan model which describes the properties and control structure of the plans the agent may employ.

A further methodology, *INGENIAS* [Pavón et al., 2008], provides a graphical agent-oriented language for the specification of social simulation models, and for transforming these models to code for an agent-based simulation toolkit. It also defines a specific environment viewpoint model for capturing the entities the MAS is interacting with, contains the resources required by tasks, assigned to agents or groups in the system, external agents whit whom it is possible to communicate, and applications. An API provides access to perceptions and actions of the agents, by producing events and by acting on the environment at invocation of their methods, and thus helps to develop MAS that will coexist with other, non-agent based applications.

The agent's environment

In most systems, a global representation of the surroundings of the whole MAS is needed, which would not only be the medium where agents act, but also a medium for sharing information and mediating coordination among agents. An important functionality of the environment should be to embed resources and services, which are typically situated in a physical structure [Weyns et al., 2007]. Several AOSE methodologies allow to model environmental concepts. However, environment modelling is often not explicit, in the sense that it consists only of high-level organization of agents or of a set of general resources.

The *Agents and Artifacts* metamodel [Omicini et al., 2006] proposes *artifacts*, which are defined as non-intentional and non-autonomous computational entities, to represent resources or tools that agents can dynamically use, share and manipulate. Artifacts operate in a transparent way, to serve agents, which can call the operations artifacts provide, to sense and to effect on the environment. Artifacts encapsulate and provide access to resources in the environment, like objects following the object-oriented paradigm. Although, they differ, because they persist in the system and exhibit an interface to available operations, invokable through messaging.

Artifacts are independent computational entities which represent any kind of resource or tool that agents can dynamically use, share and manipulate. Using artifacts, it is possible to model the environment and the mediated access to it. Agents can call the operations an artifact exhibits on its usage interface, to change their state, to get information (to sense), or to produce a desired effect on the environment. Moreover, artifacts can rely on other artifacts and exhibit a reactive behaviour on environment changes. The environment is the composition of the artifacts in a system, each with its own state.

2.4 Work Context

The aim of our work is to define a framework for the modelling and implementation of self-adaptive systems. In this section, some works will be presented, which constitute the basis upon which we will build our approach. We present the agent-oriented software engineering methodology *Tropos*, which was selected as the baseline for the framework presented in this thesis, the BDI agent platform *Jadex*, and the methodology ADELFE, which is used in an approach presented in Chapter 6.

2.4.1 Tropos

The agent-oriented methodology *Tropos* [Castro et al., 2002, Bresciani et al., 2004a, Penserini et al., 2007b] is a general development methodology which covers the whole development process, giving a crucial role to early requirements analysis, for capturing the organisational settings where a system will be embedded in. *Tropos* borrows the modelling language and analysis techniques from *i** [Yu, 1995] and integrates them with an agent-oriented paradigm.

The concept of *Agent* and related mental notions such as *goals*, *plans* and *dependencies* are used through the development phases, from to architectural design, starting from requirements analysis, modelling the stakeholders and their needs, and leading to an architectural design of a multi-agent system.

The *Tropos* development process is organized into five phases: *Early Requirements*, whose objective is to produce a model of the environment (i.e. the organizational settings) "as-is", with the stakeholders and the dependencies among themselves; *Late Requirements*, in which the system-to-be is introduced in the domain, and its requirements are defined by the stakeholders, delegating their needs to the system; *Architectural Design*, whose objective is to obtain a representation of the internal architecture of the system in terms of subcomponents of the system and relationships among them; *Detailed Design*, which is concerned with the detailed specification of the capabilities and interactions in the system; *Implementation*, whose objective is the production of code from the detailed design specification. In particular, our work takes as baseline the *Tropos* modelling process defined in [Penserini et al., 2007b].

A core activity along the *Tropos* process is conceptual modelling. The system is depicted in terms of actor models (i.e. *i** strategic dependency models) representing the dependencies between actors, including software systems, in an organisation and goal models (i.e. *i** strategic rationale models), representing the rationale of an actor's behaviour. The central concepts of the graphical modelling language are depicted in Figure 2.1. *Goals* (i.e. Hardgoals in this thesis, unless specified differently) are objectives the system under consideration should achieve, while *Plans* are tasks to be performed to achieve a goal. Following [Penserini et al., 2007b], *Softgoals* represent mainly quality of service requirements and preferences.

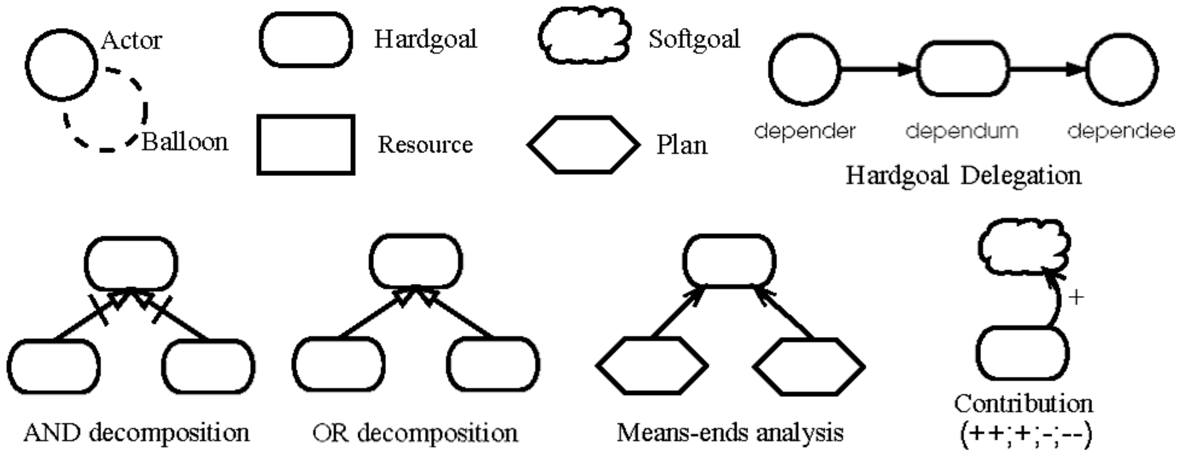


Figure 2.1: Concepts and relationships in the *Tropos* modelling language and their graphical representation.

A goal model in *Tropos* (an example is depicted in Figure 2.2 B) is represented in terms of a forest of AND/OR-decomposed goals Additionally a goal model contains means-ends relationships among plans (the means) and goals (the end), to define the means to satisfy a goal. Multiple means-ends relationships have to be seen as alternatives for goal satisfaction. Positive (+,++) and negative (−,−−) contributions from goals and plans to softgoals can also be specified. As crucial difference to *i**, the *Tropos*

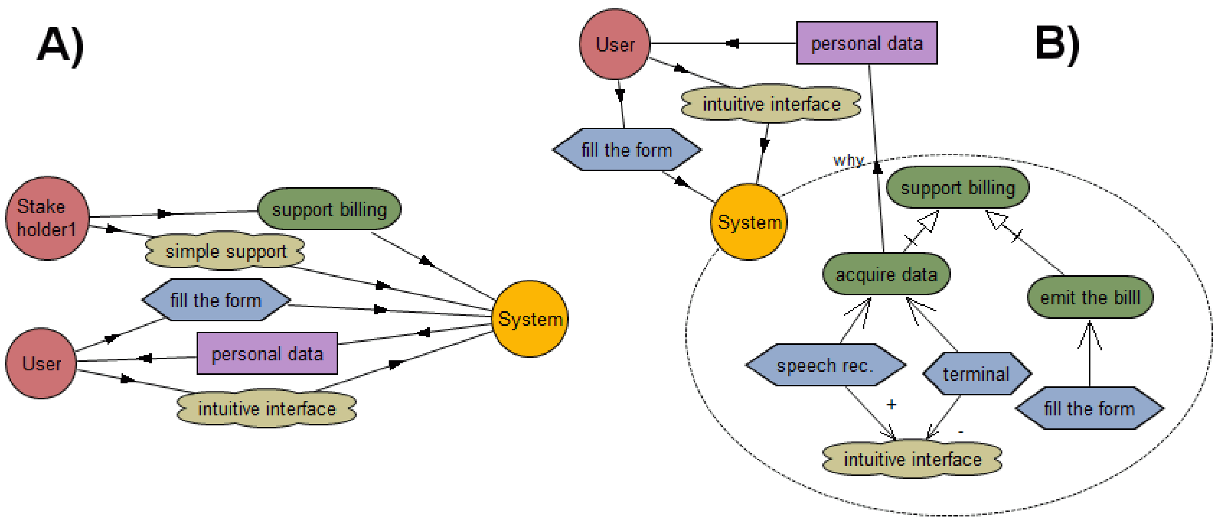


Figure 2.2: Tropos models: A) actor diagram, B) goal diagram for the "System" actor.

metamodel (see Figure 2.3) explicitly defines goal decomposition into subgoals.

In late requirements analysis, the goals are delegated from the stakeholders to the system, decomposed and analysed. The notions of actor, goal, plan and dependency are then also used to define the system architecture, where the high-level goals of the system are delegated to specialized sub-actors. Various architectural styles to guide this decomposition were proposed in literature, e.g. [Kolp et al., 2001, Bresciani et al., 2004b].

In the single sub-actors, the delegated goals are refined by goal decomposition and eventually delegated to other actors or operationalised defining the activities (i.e. the plans) to carry out. In the detailed design phase, these activities are detailed, e.g. following capability modelling [Penserini et al., 2006c], using UML diagrams and defining interaction protocols. For the implementation, *Tropos* claims to be general. If an agent-oriented implementation is chosen, the identified sub-actors will be the building blocks of a multi-agent system in the implementation phase, e.g., in [Bresciani et al., 2004a] a sketch for an implementation by JACK agents is given.

Various model analysis techniques, e.g. [Giorgini et al., 2005b], and supporting tools, e.g. TAOM4E [3] [Perini and Susi, 2004] are provided. TAOM4E is a conceptual modelling tool developed at FBK Irst, which is extended for the purposes of this thesis, as shown in Section 4.3.1.

[3]Tool for Agent Oriented visual Modelling for the Eclipse platform (`http://selab.fbk.eu/taom`).

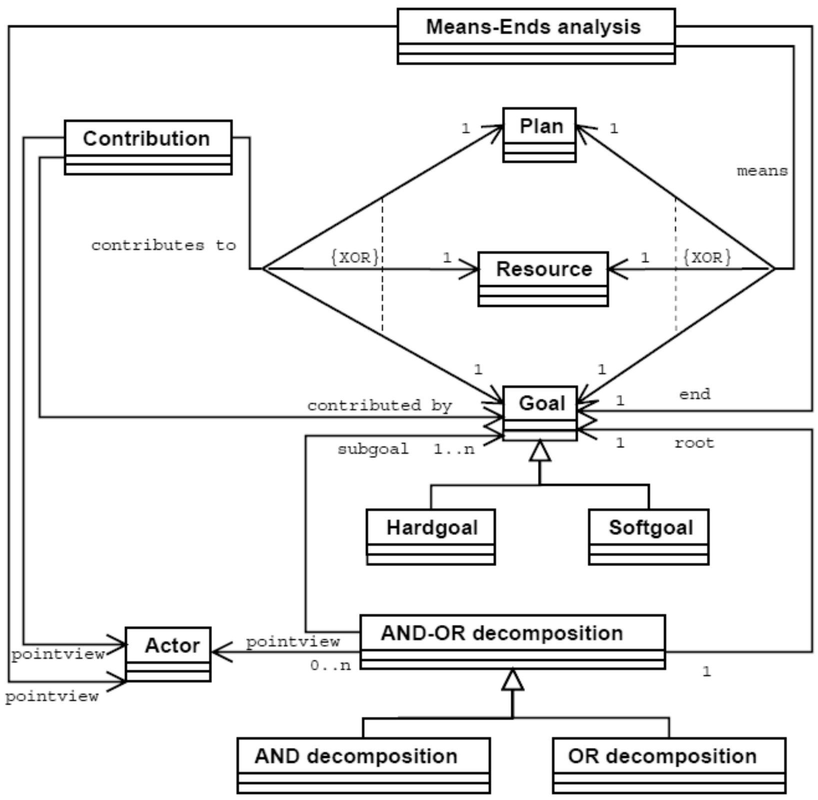


Figure 2.3: View on the *Tropos* metamodel with concepts related to the goal model (from [Bresciani et al., 2004a]).

Formal Tropos

The formal language *Formal Tropos* [Fuxman et al., 2001] gives a formal description to *Tropos* goal models. It provides a textual notation for these models and allows describing dynamic constraints among different elements in a first order, linear-time temporal logic (LTL), to define the allowed states for a system. Goals can have different types (achieve, maintain, achieve and maintain, avoid), which are defined by temporal formulas, and pre- and post-conditions can be defined. *Formal Tropos* defines precise semantics for goal models, with a different purpose with respect to our work, aiming to a formal verification of requirements models (by model checking), i.e. at finding counterexamples in a specification, for which a goal cannot be achieved.

2.4.2 Jadex

In this thesis we adopt the Jadex BDI agent platform [Pokahr et al., 2003, Braubach et al., 2004]. Jadex consists of an agent platform, a Java API and development tools. It addresses the limitations of other BDI agent programming platforms by introducing an explicit representation of goals and a goal deliberation mechanism. Jadex agents are declared in an *agent definition file* (ADF) in XML format, where the agent is specified by declaring goals with their type and conditions, a belief base, representing Java objects database-like, and available plans linked to classes implemented in Java.

At run-time, Jadex goal satisfaction is linked to a life cycle with different states. Transitions between these states are guided and guarded by several types of boolean conditions on values in the agent's belief base. The general life-cycle, represented in Figure 2.4, is instantiated for various types of goals: goals to achieve some state in the belief base (*achieve-goals*), to maintain a certain state in time (*maintain-goals*) or to execute at least one of the plans available for the satisfaction of the goal, with success (*perform-goals*). In the ADF, various options for the goal satisfaction dynamics can be defined.

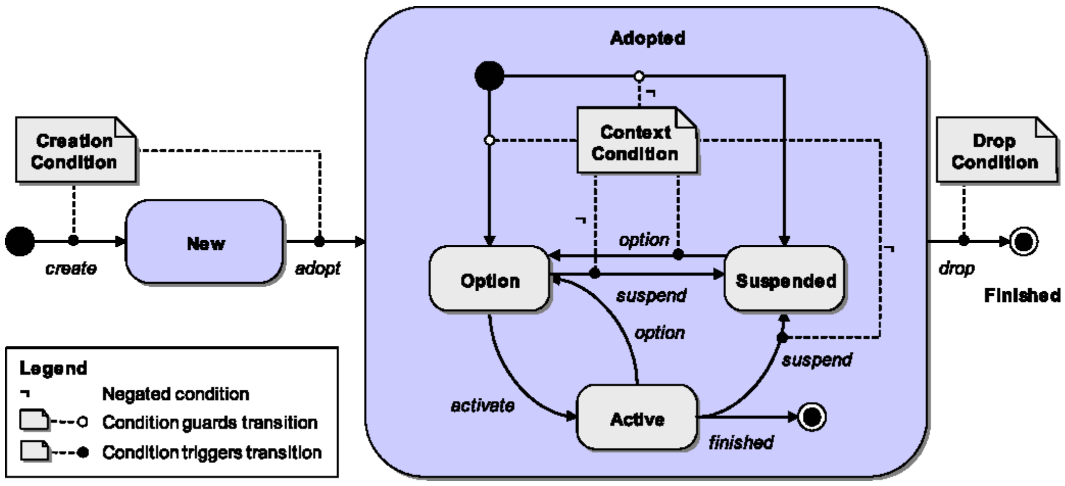


Figure 2.4: General Jadex goal life-cycle (from [Pokahr et al., 2005]).

For the plans, the ADF contains the interfaces with input and output parameters, the goals that can be achieved by a plan and a reference to the Java file containing the corresponding code. In short, if a goal is activated, one of the plans (i.e. the corresponding Java class) defined as a means to achieve it, will be triggered for execution. The belief base can be defined and populated in the ADF. It can be composed by beliefs which are stored locally, and *belief sets* referencing to Java classes. Moreover, the ADF contains the initial state of the agent, a definition of the possible messages the agent can send and receive, and SQL-like queries, which can used as a shorthand to retrieve facts from the belief base.

In the Jadex plans, implemented in Java, the whole agent definition can be accessed through an API, e.g. for reading and modifying facts in the belief base, for adopting goals, and also for run-time modification of the goals defined.

Recently, a completely new version of Jadex was made available, which is realized on top of a standard rule engine. Since the new release is not backwards-compatible and still in beta stadium (January 2011), our work relies on the latest stable version of Jadex (version 0.96).

2.4.3 ADELFE

ADELFE [4] [Bernon et al., 2005] is an agent-oriented methodology for designing Adaptive Multi-Agent Systems (AMAS). Systems developed according to ADELFE consist of *cooperative* agents which provide an emergent global behaviour not explicitly coded inside the single agents. Such systems follow the AMAS theory [Capera et al., 2003], giving to the single agents the ability to autonomously and locally modify their interactions with their peers, in order to adapt to changes in their environment. Adaptation is thus realised by self-organisation of the single agents. These agents pursue their individual objective, acting cooperatively, and trying to avoid any *Non-Cooperative Situation* like conflict, unproductiveness, or concurrence, properly acting to come back to a cooperative state (e.g. by changing cooperation partners, changing protocols, or adapting bid offers). Thus, the modelling process follows a *bottom-up approach*, defining the cooperation rules and activities of the single agents, leading to an emergent behaviour of the system.

The ADELFE methodology covers the software design phases from the requirements to the implementation, with the addition of specific activities to support the design of adaptive multi-agent systems. It is based on the Rational Unified Process (RUP) and extends it along various lines. After definig the user requirements in terms of goals in *Preliminary Requirements Modelling*, in *Final Requirements Modelling* the environment is characterised by determining the domain entities and their environmental context. In the *Analysis* phase, ADELFE extends RUP by steps for verifying the adequacy of the problem to a solution with an AMAS, identifying the cooperative agents among the entities in the domain, and determining the relationships between these agents.

In the *Design* phase, the single agents are detailed, characterising all the concepts involved, including their perceptions, actions, skills, aptitudes, knowledge representa-

[4] ADELFE is a French acronym for "Atelier de Developpement de Logiciels a Fonctionnalité Emergente", see `http://www.irit.fr/ADELFE`

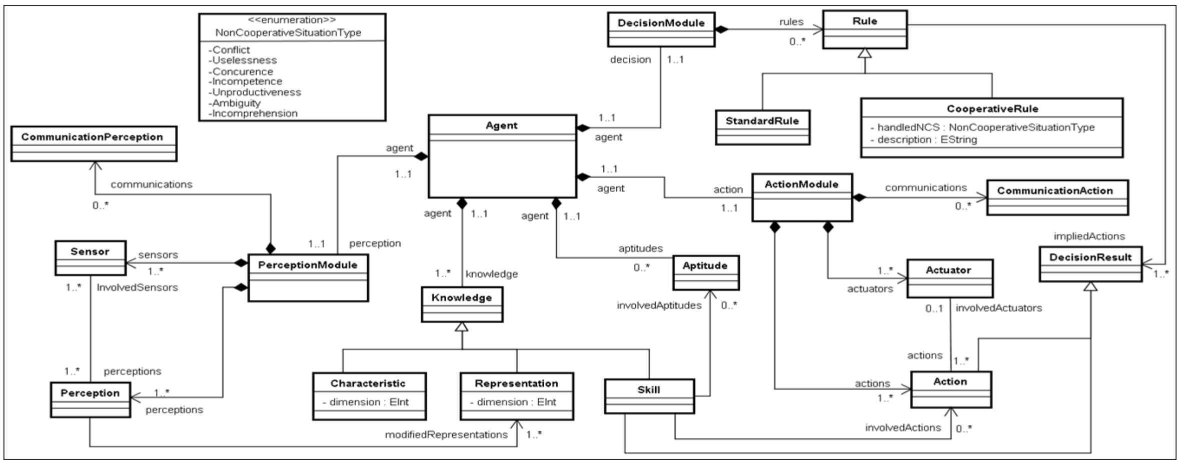


Figure 2.5: Portion of the ADELFE metamodel concerning a cooperative agent (from [Rougemaille, 2008]).

tion and interaction languages (refer to the metamodel in Figure 2.5 for details on the available concepts). Note, that there are no direct relationships between the high-level goals and the agent's detailed design, in this phase. Starting from the possible interactions, Non-Cooperative Situations (NCS) are identified. Rules to detect NCS and methods to recover to re-establish a cooperative behaviour, have to be defined.

While the original ADELFE uses UML for a graphical visualisation, we adopt the domain-specific AMAS Modelling Language (AMAS-ML) [Rougemaille, 2008]. AMAS-ML provides a system-environment diagram showing the participating entities and the cooperative agents in the domain (Figure 6.4), an *agent diagram* defining the agent's internals in the form of skills, aptitudes ad represented beliefs, and a behavioural rules diagram defining rules to recover from NCS.

Adelfe provides a model-driven trasformation of the design models to a specific agent definition language, which is finally mapped to a Java project implemeting a prototype of the system.

Chapter 3

Extending Goal Modelling for Adaptivity

3.1 Introduction

Software needs to have alternative ways to satisfy its requirements, to properly adapt its behaviour in a dynamic, changing environment.

Despite goal models allow designers to capture these alternative ways to approach a problem, the only use of this design approach is not sufficient to support and improve the decision making mechanisms within intelligent systems such as the autonomous selection of alternatives. In fact, self-adaptive systems should not only know multiple alternative ways to approach a problem, but also to be able to decide autonomously at run-time which alternatives to pursue and when to change their behaviour. Thus, to carry out decisions, the system needs to have access to additional knowledge.

Our objective is to provide to a software system at design-time the acquaintance needed in order to increase its ability in interpreting contextual information and taking congruent decisions at run-time (i.e., self-* properties), to meet its requirements in a dynamic environment.

Starting from the agent-oriented software engineering methodology *Tropos* (see Section 2.4.1), we define a development framework called ***Tropos4AS*** (***Tropos for self-Adaptive Systems***), extending *Tropos* along different lines to better support the modelling, design and implementation of self-adaptive systems.

Tropos4AS includes conceptual models, a graphical language, guidelines, and supporting tools, in order to model systems with adaptive properties using the concepts of goal, system environment and failure recovery. The adaptive properties a system is expected to exhibit will be concretised step by step, leading to an implementation of a self-adaptive system by *adaptive agents* based on a BDI architecture. To catch crucial requirements of a self-adaptive system, the framework bases on three modelling pillars:

- a **goal model** based design indicating goals and relationships the system must adhere to,
- an **environment model** representing the key elements in the context of the system, which affect the achievement of goals, and
- a **failure model** that supports designers in capturing unwanted states of affair and possible recovery procedures either to anticipate foreseeable failures or to recover from unpredictable ones.

A mapping to an implementation which contains a run-time representation of these models is presented in the subsequent chapter.

We aim at capturing **at design time** all the available information about possibly changing goals and about the dynamics of goal achievement. Giving also the possibility to reason on its goals **at run-time**, the system is able to take decisions on the behaviour to exhibit, to respect the requirements. The definition of architectures and run-time techniques to increase the domain knowledge, to forecast situations or to fine-tune its behaviour (e.g. learning techniques, heuristics, or statistical analysis) is not an objective of this work, but would indeed be complementary to our approach.

Outline This chapter gives a comprehensive and detailed description of the *Tropos4AS* framework, introducing various extensions to the *Tropos* modelling language and to the modelling process, with the aim of giving to the software a more precise specification of the variability and the decision making process.

In this chapter, in Section 3.2, we present and motivate the single extensions to the modelling language in detail, for the specification of details for goal deliberation and achievement, along with their relationships with the environment, and with possible failures, errors causing them, and corresponding recovery activities. The meta-models of the modelling language, the intuitive semantics of the new constructs and their graphical notation are defined.

The modelling process described in Section 3.3 adds various modelling steps for the newly introduced extensions to the architectural design (AD) phase of *Tropos* as defined in [Penserini et al., 2007b] (see Section 2.4.1 for details).

3.2 Conceptual Models

The *Tropos4AS* framework is built taking as baseline the conceptual models and modelling process of the *Tropos* methodology, as defined in [Penserini et al., 2007b]. This work uses *Tropos* explicitly for capturing variability in the requirements and divides goal modelling into two design abstraction levels that characterize the whole development of a software system: the *knowledge level* and the *capability level*, as depicted on an example in Figure 3.1.

The ***knowledge level*** includes the components in charge of taking decisions about which behaviour to exhibit in a specific situation and to plan appropriate activities to carry out, that is, the goals and their relationships with other entities in a goal model.

The ***capability level*** brings about the functional, executable parts of a system, which are the means to bring about the system's leaf goals.

Our work focusses on the *knowledge level* of a software system. The capability level definition regarding the detailed design and code generation of the system's functionalities, as realised e.g. in [Penserini et al., 2006c] by *Tropos* along with UML activity and sequence diagrams, is not in the scope of the present work.

Tropos4AS extends *Tropos* goal models along three lines: environment modelling (Section 3.2.1), goal modelling (Section 3.2.2) and failure modelling for goal failure prevention (Section 3.2.3). Starting from the *Tropos* goal metamodels defined in [Susi et al., 2005], we present extended metamodels and give intuitive semantics for the newly introduced modelling concepts.

Aiming at a detailed definition of the single components or *agents*[1] in the system, our extensions refer specifically to *Tropos* goal models at the *Architectural Design (AD)* phase. In this phase of *Tropos* the abstract system-to-be in output from the *Late Requirements* phase is divided into software agents that, together, realise the software system. The *Tropos4AS* process extensions are incorporated in this phase, when the delegated goals are decomposed, detailed and operationalised.

The *Tropos4AS* metamodel, depicted with different highlights in the Figures 3.2, 3.3, and 3.6, defines the concepts introduced to characterise a self-adaptive system.

The metamodel builds upon the central concept of ***system actor*** of a goal model in the *Tropos* AD phase, which represents the system-to-be, along with the **sub-system *agents*** identified (specifically, step 7 in [Penserini et al., 2007b]), which will be in charge of realising the system's objectives.

[1]in the context of *Tropos4AS* modelling, from now on we will refer to the single components (actors) in a system, which are goal-directed and autonomous, as *agents*.

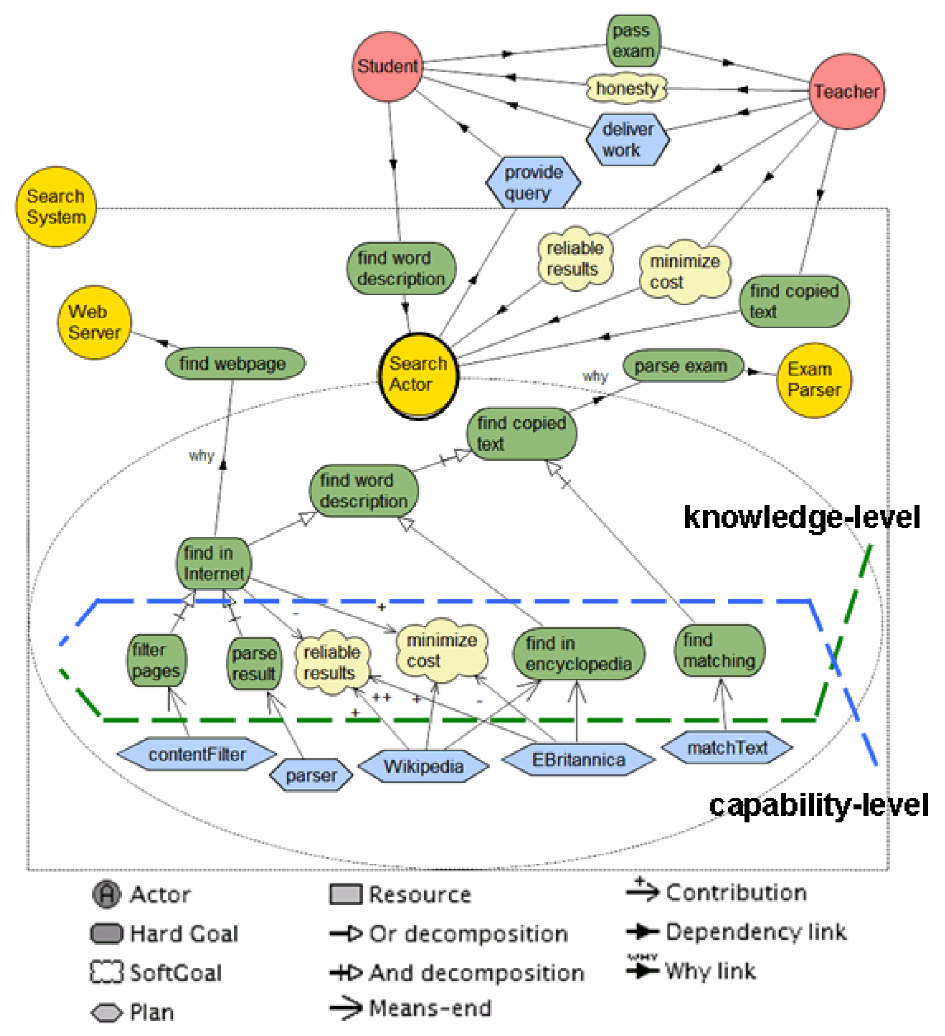


Figure 3.1: Knowledge and capability level in a Tropos goal model.

In the following, we describe the new modelling extensions, whereas in Section 3.3, the design process for creating the models is given.

3.2.1 The environment model

Self-adaptivity is concerned with *how* a software system behaves when substantial changes in its environment occur. Thus it is essential that the system can percept its environment, to relate changed circumstances to appropriate system behaviours. As often literature reports about adaptivity in AI (e.g., see Ch. 3 in [Sutton and Barto, 1998]), learning abilities and decision-making strategies of an agent strictly depend on the observation of environment changes. While in [Sutton and Barto, 1998] and related approaches researchers consider the agents and their environment as black-boxes, focussing on a formal definition of the interaction between them, we propose a way to make explicit the inside of an agent's decision-making and its dependencies with the environment.

Tropos4AS allows to model the key relationships between agents' goals and the entities situated in their environment. Following the *Tropos* requirements analysis phase, possibly involved intentional entities[2], i.e. the actors, both in the system to develop and in the surrounding social context, are captured in actor diagrams (*i** Strategic Dependency models [Yu, 1995]).

The *environment model* captures the *non-intentional* entities necessary for interfacing the system with the surrounding world. This model will be kept simple deliberately, limiting only to the relationships between environmental entities and the agent. It is not our aim to define a conceptual model of the whole domain, to capture the various relationships between the entities, or to detail their internals. Available modelling languages, from UML class diagrams to domain ontologies, can be used for this purpose, depending on the needs and the extent of a project.

We define the non-intentional entities following the idea of *Artifact*, proposed by Omicini et al. [Omicini et al., 2006] in the *Agents and Artifacts* meta-model. As opposed to agents, artifacts are entities without an autonomous, proactive behaviour (or, however, they are perceived as such from the agents' perspective). Rather, they are passive entities used by the agents to sense or act in the environment or to hold data for interchange or for persistent storage. They provide *functionalities* through usage interfaces, to access the environment (for sensing or acting), and properties describing their internal state. Thus, artifacts are *inspectable* [Molesini et al., 2005], while agents have a mental state not accessible to other agents in the environment [Rao and Georgeff, 1995].

Working on the *knowledge level*, not on the agent's single capabilities (the *capability level*), we are mainly interested in the functionalities used to sense in the environment, in order to improve the decision making process of the agent, e.g. making the agent more aware about which behaviour to adopt to deal with a contingency.

An artifact (Figure 3.2) can be a physical entity or also some software service, and

[2]We distinguish between intentional and non-intentional entities. *Intentional* entities are the actors in the system (stakeholders and other human and software actors), acting goal-oriented, pro-active and with some degree of autonomy, while *non-intentional* entities are all the remaining passive, function-oriented entities, designed to encapsulate some kind of function [Omicini et al., 2006].

can be internal or external to the system-to-be, and is thus either under the control of the system developer, or situated outside of the system boundary, in the external world, from the viewpoint of the system.

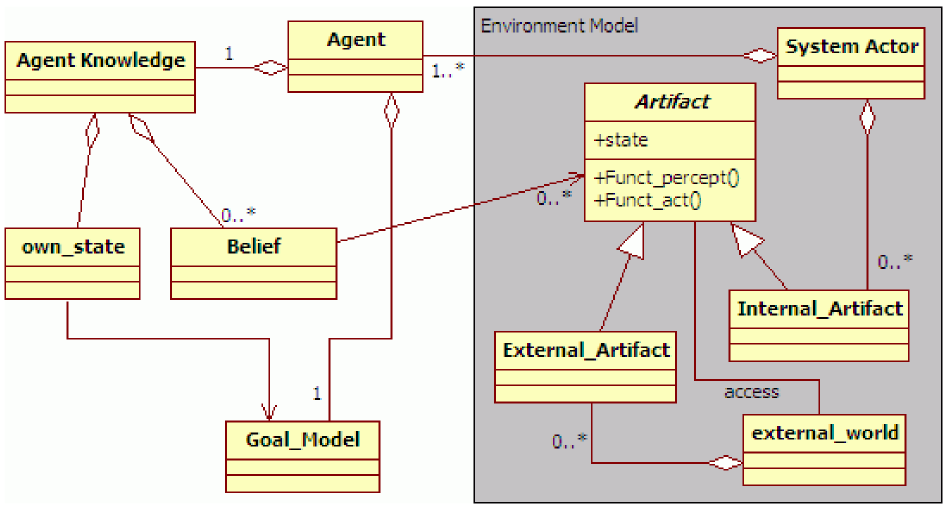


Figure 3.2: Detail of the *Tropos4AS* meta-model, focussing on the relationships of the agent's knowledge with the surrounding environment, represented by artifacts.

The boundary between an agent system and its external world is not strictly related to the physical boundary of a robot or a machine. Rather, this boundary depends on the purposes of modelling. For the purpose of modelling the knowledge of an agent, anything under the control of the developer (including artifacts that offer an interface allowing agents to sense and act in this environment), can be considered as internal to the system, whereas anything that is not under control of the developer is considered to be part of the external world. In a cleaner robot example, the various sensors, the broom, the floor and the dustbins are non-intentional artifacts in the environment: the former two (sensors, broom) being inside the system, the latter two (floor, dustbins) being part of the external world.

The concept of *Agent* (depicted in a fragment of the meta-model, in Figure 3.2) contains a goal model, the agent's knowledge represented by beliefs related to data sensed by artifacts, and the agent's own state at run-time, representing the actual goal satisfaction life-cycle and its achievement state.

3.2.2 The Extended Goal Model

Using *Tropos* as the baseline along all the approach, we try to capture at design time the requirements essential for decision making at run-time.

To capture details of interest for decision-making in a self-adaptive system, we extend the goal model adding information regarding dynamic aspects of goal creation and achievement. This is achieved by the use of conditions related to the agent's knowledge (i.e. its awareness about the environment), that guide or guard goal achievement at run-time. Figure 3.3 displays an excerpt of the meta-model describing the possible goal types, along with their available conditions, which are detailed in the reminder of this section.

Softgoals

A first criteria for alternatives selection at design time is provided by *Tropos* softgoal contributions modelling. We use positive and negative contributions to softgoals (representing non-functional requirements, e.g. users' preferences and needs, and quality of service (QoS)), such as qualitative measures, e.g., '+', '++', '–', '– –' as well as quantitative measures (expressed by numerical values or functions, as e.g. in [Lapouchnian et al., 2006]) for guiding the selection between possible alternatives at run-time.

We **extend softgoals** by adding an additional information: the ***importance*** that is attributed to a softgoal at a given moment at run-time. For our purposes, we selected to capture importance by a value in an interval from 0 "unimportant" to 1 "very important".

Giving more or less importance to the various softgoals modelled, the selection of alternatives can be influenced by the stakeholder or directly by the user, at run-time. The algorithm for softgoal contribution maximisation at run-time has thus also to consider softgoal importance. Contributions to softgoals that have no importance, should be ininfluential to alternatives selection, while e.g. a positive contribution to a softgoal with an importance of 1 should give a higher opportunity for selection than one to a softgoal with an importance of 0.5. Our prototype tool for a mapping to the implementation, ***t2x*** (Section 4.3.2), provides a simple contribution maximisation algorithm that considers also softgoal importance.

Goal types

In *Tropos*, goals denote a state of affairs to be achieved [Bresciani et al., 2004a]. To effectively deal with the modelling of the pro-active behaviour of a system, we need to understand what it means for the system itself at run-time to bring about its goals.

The necessity to achieve a goal can depend on other goals' achievement or on environmental circumstances, which may change also during goal achievement. It can be necessary to achieve goals once, or to maintain some state in time. Moreover, it can be needed to suspend goal achievement, e.g. to reach another, conflicting goal which is necessary in some circumstance; and to drop goals, if their achievement is no more needed or considered to be impossible in some state. All this information is critical if we want to model and finally implement systems aware of their environment and of the possibilities they have for adaptation to their context to achieve their high-level goals.

Tropos4AS focusses on the semantics of an agent's goal model (i.e., in the later design phases of *Tropos*) mainly driven by adaptivity features and target programming languages that allow for a concrete implementation. This design choice is, in a different fashion, also supported at run-time by most BDI agent programming languages, e.g. Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008], and Jack [Howden et al., 2001].

Tropos4AS introduces three goal types to define the the attitude of an agent towards goal satisfaction at run-time: *achieve-goals*, *maintain-goals*, and *perform-goals*, also proposed in the work of Dastani et. al [Dastani et al., 2006].

Different goal types were already presented and formalised in the *Formal Tropos* language[3], which was however developed with the aim of consistency verification via model-checking techniques and has thus several strong limitations for the activation, the re-activation and the dropping of goals, to limit the explosion of the state space for the formal analysis. Similarly, also the KAOS [Dardenne et al., 1993] formal specification language defines goal types by temporal logics. On the contrary, *Tropos4AS* focusses on the semantics of an agent's goal model (i.e., in the later design phases of *Tropos*) mainly driven by adaptivity features and target programming languages that allow for a concrete implementation. This design choice is, in a different fashion, also supported at run-time by most BDI agent programming languages, e.g. Jadex [Pokahr et al., 2005], 2APL [Dastani, 2008], and Jack [Howden et al., 2001].

To characterise these goal types, we need to understand the process of goal achievement at run-time, with the aim of being able to define goals that are achieved e.g. once in a certain situation, during a period of time, or simply by processing specific activities with success. For this purpose, we introduce a simple model for the states in a goal's achievement process. These states are defined in various ways in existing agent implementation platforms. Riemsdijk et al. [van Riemsdijk et al., 2008] identified a simplified set of states, as a common denominator for capturing most goal types available in the prominent BDI agent programming languages. We adapted this minimal definition, represented in Figure 3.4, to explain the concepts introduced in this chapter,

[3]*Formal Tropos* is based on linear time temporal logic (LTL), see [Fuxman et al., 2001].

for single goals. Later, in Chapter 5, we introduce a formal, more detailed definition of the introduced goal types, in the context of an AND/OR goal decomposition tree.

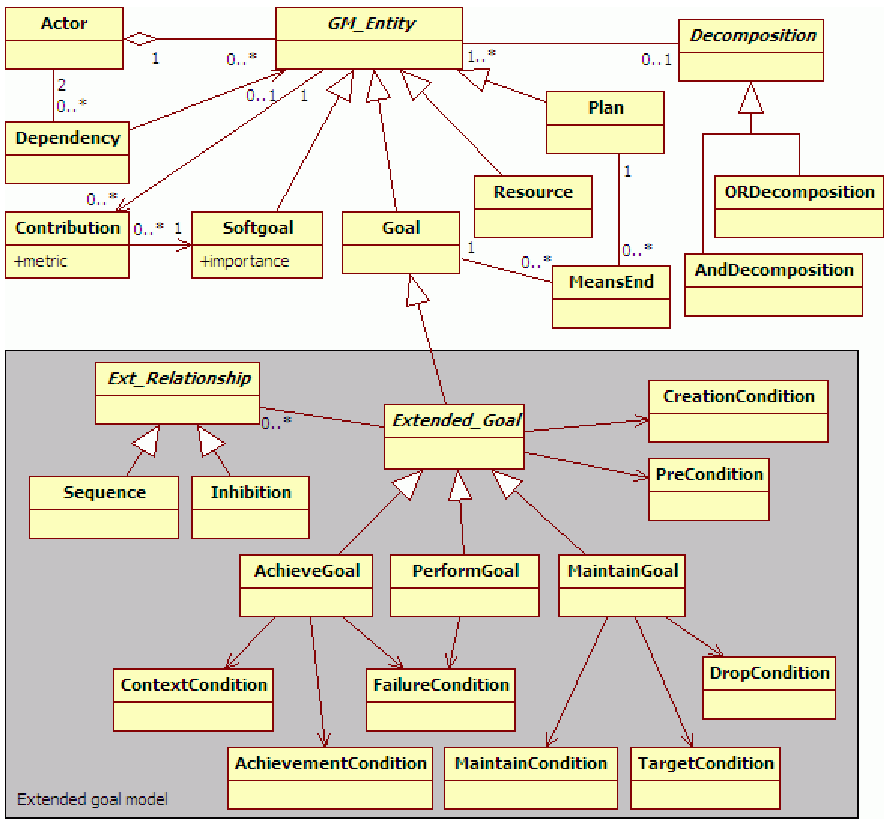


Figure 3.3: View on the *Tropos4AS* meta-model showing the central parts of the *Tropos* goal model as defined in [Susi et al., 2005], and the extended goal concept, with goal types and related conditions.

Referring to Figure 3.4, goals can be in the following states: they can be either **not adopted** or **adopted**, whereas adopted goals can be either **active** or **suspended**. The above-mentioned goal types can be modelled by defining the transitions between these states. In the following, they are explained by giving an intuitive description of these state transitions, as in Figure 3.5. Each goal type is associated to a set of conditions (see Figure 3.3), which define, together with other events, its run-time behaviour in response to environment changes, by guiding or guarding state transitions in the goal satisfaction process. The conditions are detailed in the subsequent section.

Achieve-goals are characterized by an achievement condition that specifies when a certain state of affairs is reached. The satisfaction of the goal can be attempted

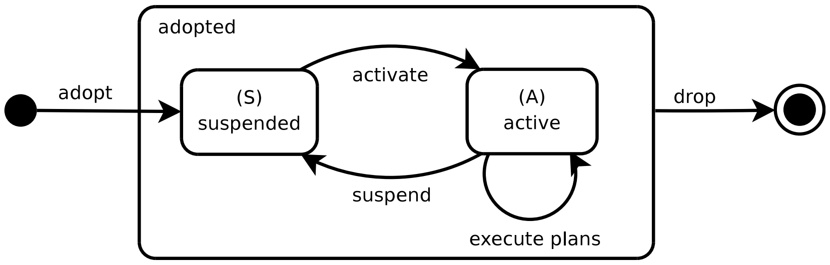


Figure 3.4: Possible states of a goal at run-time (after [van Riemsdijk et al., 2008]).

several times till this condition holds. Moreover, a failure condition can terminate goal achievement, defining it as failed.

Perform-goals are satisfied by successfully executing one of the associated activities (i.e. *plans* or *goals*). Success or failure are reported, without evaluating a particular achievement condition.

Maintain-goals try to maintain a certain state of affairs. In literature, different types of semantics have been attributed to maintain-goals. An agent can be reactive or proactive in maintaining a state. In the first case (*reactive* maintain-goals), it starts taking action when a particular state is no longer maintained, while in the second case (*proactive* maintain-goals) it has to act in anticipation to *prevent* the failure of the maintenance condition (see also [Duff et al., 2006]). In this work we focus on *reactive* maintain-goals, which are available on most agent platforms. Such goals are *activated* each time their maintenance condition is not satisfied and *suspended* if a target condition holds.

In comparison, the *Formal Tropos* language adopts the conceptually different *proactive* maintain-goals, which are suitable for formal verification (supposing that the state of the system can be defined completely, without unexpected changes). Requiring predictive reasoning mechanisms, such goals are not easily representable in procedural agent languages in general [van Riemsdijk et al., 2008], and approaches such as a "look-ahead with rollback" [Hindriks and van Riemsdijk, 2007] are deployable only in specific domains. However, supposing, like e.g. in [Thangarajah et al., 2010], to have reliable prediction mechanisms, the semantics for proactive maintain-goals will again correspond to reactive ones.

The expected behaviour exhibited by the agent for the satisfaction of these three types of goal inside a goal model, along with the different conditions that determine their

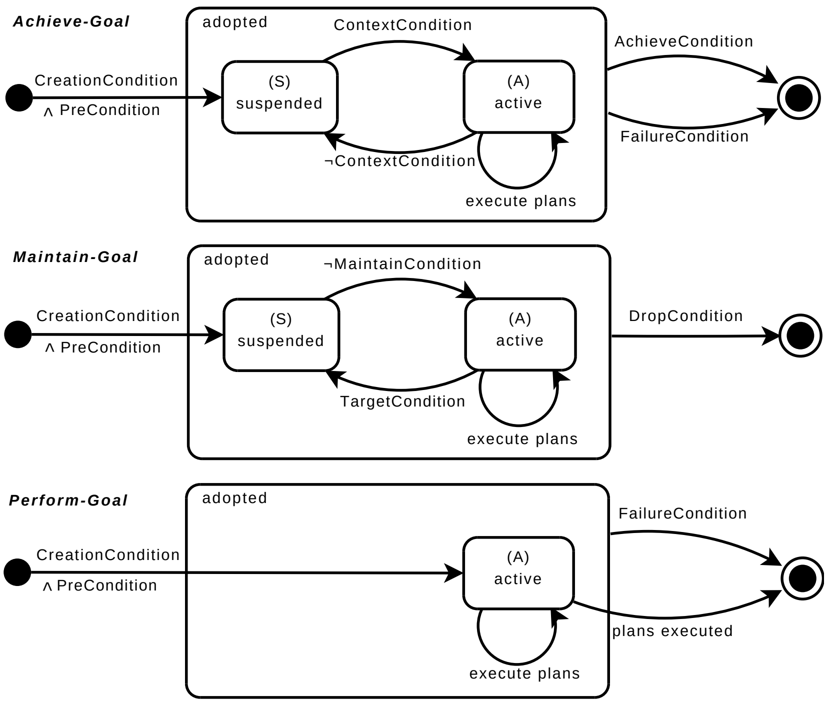


Figure 3.5: State diagrams for the goal satisfaction process, adapted for the three basic goal types achieve-goal, maintain-goal and perform goal, with associated conditions.

life-cycle, founds on a formal model described in Chapter 5.

Goal conditions on environmental states

A self-adaptive system is by definition strongly related to the environment in which it is situated. *Tropos* goal models are able to capture relationships between different actors in a system at an organisational level, by modelling dependencies. Having the system's goal model and a model of its environment, also the relationships between environmental artifacts and goals in the goal model can be captured.

To concretise the relationships between goals and the system's environment, we introduce the concept of *condition*, a choice well supported at run-time by various agent programming languages.

A *goal condition* (see Figure 3.6) relates the satisfaction process of a goal to the agent's knowledge, which comprises the agent's own state (e.g. the goal model achievement state) and the access to environmental artifacts, their state and their perception functionalities. This is achieved by defining a boolean expression evaluated on the agent's knowledge. For example, the state of an environmental artifact (e.g. a value reported by the 'dirt sensor' $S2$ of the cleaner robot CleanerSystem in Figure 7.4) can be related to the achievement of a goal OfficeClean by a condition $dirtlevel \leq threshold$.

The types of conditions introduced in *Tropos4AS* are associated to the goal types previously introduced, as defined in Figure 3.3. They guard transitions between goal states or guide the goal achievement process, triggering the transitions between the states.

The expressions can be defined, depending on the aim of the modeller, the purpose and the granularity of the model, by natural language, or as a boolean expression on the properties and functionalities delivered by the artifacts or on the methods and fields provided (e.g. in JAVA) to access them. In the following, the condition types and their effect on the goal achievement process, shown graphically in Figure 3.5, are described.

Creation conditions (CC) determine the criteria for adopting a goal and thus starting the goal satisfaction process, transiting, depending on the goal type, from the *non-adopted* to the *suspended* or *active* state.

Pre-conditions are guarding conditions that have to be fulfilled to adopt a goal. Moreover, pre-conditions can also be used as a guarding condition for the applicability of a plan, i.e. to define the environmental state in which a plan is available to be executed to reach a goal. Since preconditions are evaluated in conjunction with creation conditions, they could in practice be integrated into them. However, often pre-conditions are needed in addition to the implicit creation of a goal (e.g. decomposition relations, dependencies,...).

Context conditions (CoC) have to be valid during goal achievement. A goal in *suspended* state transits to the *active* state if the condition is satisfied. As long as the condition is not satisfied, the goal is suspended.

Achievement conditions (AC) define when an *adopted* achieve-goal was successfully achieved and is thus dropped. Note that the name of a goal is typically a (often vague or ambiguous) description of its achievement condition.

Maintain conditions (MC) and *target conditions* (TC) characterise start and end of the maintain-goal achievement process. The maintain condition denotes the desired state of a maintain-goal – a violated maintain condition brings a suspended goal to the *active* state, as the agent needs to act proactively (executing proper plans) to re-obtain this state. A target condition (which has to subsume the maintain condition) can be defined for the suspension of an active goal, to avoid a frequent reactivation. If it is not defined, the fulfilment of the maintain condition determines the transition to the *suspended* state.

Failure conditions (FC) for achieve- and perform-goals and drop conditions (DC) for maintain-goals are used to define states in which the designer considers that it is impossible to achieve a goal or no more desired to achieve it, and thus it is dropped from the adopted state. These conditions can define temporal limits for goal achievement (e.g. *room not clean at 8PM*), the need to withdraw a subgoals because of changing circumstances (e.g. if achieving an alternative goal is considered to improve the satisfaction of the requirements in a certain environment, to avoid deadlocks, etc.

In a goal model, creation conditions can also be implicit, i.e. not explicitly defined. This is the case with goals in an AND/OR decomposition tree and with goals that are the dependum part of a dependency. The adoption of such goals is typically triggered by the parent goals, or, in a dependency, by the dependent actor. Thus, defining a creation condition for children goals of a decomposition, they are therefore independent from the parents in the goal hierarchy at run-time. This can be desirable in some cases, but could have undesired effects as it does not comply to the intentional meaning of goal decomposition.

Conditions cannot only be stated for leaf level goals, but also for higher level goals in a goal hierarchy, directly affecting the decision making process of an agent on the selection of (sub)goals to achieve. While formal semantics for goal types and conditions in leaf goals, available in various agent programming languages, can be found e.g. in [van Riemsdijk et al., 2008] and [Dastani et al., 2006], Chapter 5 defines formal semantics for goal types and conditions in non-leaf goals of a goal decomposition tree.

Goal relationships

Starting with an example, if an emergency occurs, a robot has to be able to interrupt the current execution (i.e., the goal achievement process) in favour of an alternative strategy, and to resume the previous (core) activity later. To express such detailed requirements in a goal model, we endow *Tropos4AS* with additional relationships between goals, *inhibition* and *sequence*. *Inhibition* expresses run-time precedence between goals. The definition of a specific *sequence* of goals is often essential, e.g. for defining a sequential workflow among subgoals that together contribute to the achievement of their parent goal. The two relationships have the following informal semantics.

Inhibition expresses a simple form of priority between active goals. If a goal A inhibits goal a B, any time A changes to the *active* state (Figure 3.4), the achievement process of B has to be suspended until A is dropped (and, if A is a *maintain-goal*, until the desired state is reached and the goal is suspended). The relation is supported in goal-oriented programming languages such as Jadex [Pokahr et al., 2005]. We opted for the use of one-to-one relationships, to have a clear visual representation.

Sequence denotes a sequential order for the achievement of goals which are active at a given instant. It can be represented by an one-to-one relationship between two goals, or alternatively, in an AND-decomposition, by annotating the subgoals with an ascending numbering.

Inhibition can alternatively be represented by setting a context condition in the inhibited goal which is false (and thus suspends the goal) as long as the inhibitor goal is in process, while sequence (also called *prior-to* [Fuxman et al., 2001] or *precedes* [DeLoach and Miller, 2009]) can be realised by a goal precondition in the successor that the predecessor goal succeeded.

By combining *Tropos4AS* modelling concepts, additional temporal relationships can be made explicit: for example, by modelling a *Tropos* resource dependency between goals, the *depender* goal can only be achieved after the *dependee* goal reached the state denoted by the resource (the *dependum*). Goal preconditions and creation conditions give further possibilities for modelling relationships between goals. Recently, Liaskos et al [Liaskos and Mylopoulos, 2010] introduced relationships for temporal annotation into *Tropos*, namely *precedence* and *effect*, which are however already covered by our extensions. Similar causal relationships are defined by *triggers* [DeLoach and Miller, 2009] that create or drop goal when an event occurs.

3.2.3 The failure model

The extended goal model presented in the previous section gives a detailed description of the requirements of the system-to-be. Modelled alternatives give a source for adaptivity to the environment, but nevertheless the goal models typically describe “happy path” scenarios defining the default software behaviour when all works as intended.

A main feature expected from a self-adaptive system is to prevent goal failure by adapting correctly to new circumstances. However, especially in a changing, open environment, unexpected exceptional situations (e.g. due to an internal failure or due to an unexpected environmental change) could lead to degraded system performance or to failure. Often, the cause for failure is an improper use of available capabilities, a lack of capabilities, or a lack of knowledge for the composition and usage of own capabilities, to handle a specific situation. The model presented hereafter aims to give to the system engineer a means to capture, besides the goal model, also such exceptional situations and the proper way to react to them, to anticipate such failure.

A failure of the software system is in any way caused by an incompleteness of the requirements, namely, by circumstances (i.e. inputs) that were not considered at requirements analysis and design time. Notice that, for example, the physical failure of an indispensable part of a robot does not imply the failure of the control software, as long as this failure was anticipated and correctly handled.

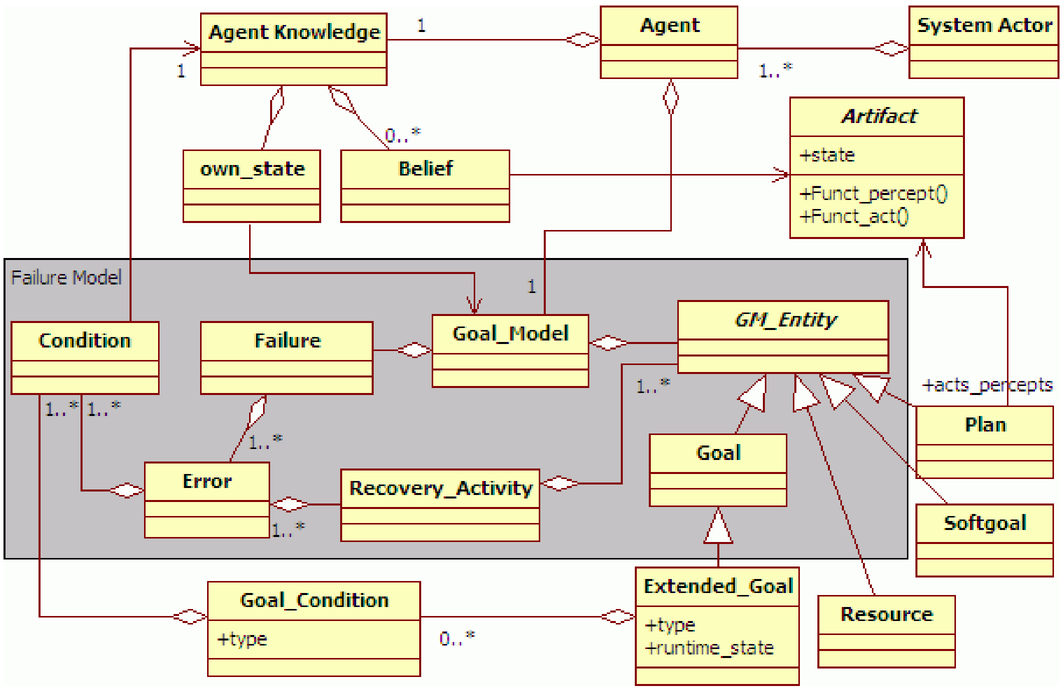


Figure 3.6: The Tropos4AS meta-model with details on conditions and failure modelling (extended goal model simplified).

It could be argued that a truly self-adaptive (or autonomic) system has to properly adapt to avoid any failure, also if the failure was not recognized at design time. The following citation by Dan Berry argues why it is not possible to build a system that is so autonomous that it is able to avoid failures or to fix errors, which have not been considered at the system's design:

> For an autonomous system to fix any fault, the humans implementing the system have to have anticipated the fault, and if a fault is anticipated, it is not a design fault [Berry et al., 2005].

This implies that systems have to be failure-proof by design, especially when considering also the run-time platform, which could have the ability to restrict the inputs or to absorb anomalies in the output of a software, to be part of the overall system's requirements and design choices.

Tropos4AS extends *Tropos* by a ***failure model***, representing failures, their causing errors, and suitable recovery activities, to complement goal models with failure prevention techniques.

The failure model explains how to act to remedy an imminent failure, to achieve a goal also when originally the agent was unable to do so by its own knowledge and capabilities. *Tropos4AS* aids the designer in anticipating failures at design time, by eliciting possible errors causing them and by analysing the possibilities to fix them.

Concepts of failure modelling In, the *Tropos* goal model, which is concerned with modelling of goals as the desired states of affair, a goal fails if it is no more possible to be achieved or no more wanted to be achieved (if this was defined by the designers, e.g. by modelling conditions). Depending on the purpose of the goal (e.g. it can be indispensable or just one of various alternatives to satisfy the stakeholders' requirements), *Tropos4AS* gives the possibility to explicitly define critical goal **Failures**, which denote the inability to reach a goal. Failures are related to the corresponding goals in the model and analysed by identifying perceivable **Errors** that may cause them. These errors, undesirable states, which are known to the designer for (possibly) leading to a failure, are defined in the form of conditions on the agent's knowledge (which includes the own state and the belief about its environment). To *anticipate failure*, for each error, various **Recovery Activities** can be defined, which provide new capabilities or a new arrangement of known capabilities, to cope with the particular situation, to *prevent* or to *recover* from the error state.

The recovery activities are (parts of) goal models, and as such modelled using the *Tropos* language. Thus, they can be considered as a homogeneous extension to the system goal model. The designer can choose to detail these goal model fragments by applying the *Tropos4AS* goal modelling activities (an example is shown in Figure 7.4). The *Tropos4AS* meta-model part in Figure 3.6 highlights these concepts. The modelling process is detailed in Section 3.3.2.

Various levels of "failure", or "reliability", can be distinguished in natural language, e.g. in English. We adhere to the terminology given by Parhami [Parhami, 1997]. Parhami defines seven levels of reliability: *Ideal, Defective, Faulty, Erroneous, Malfunctioning, Degraded and Failed.* For our purpose of modelling, we reduce this list to three essential concepts, defined in the following way:

Failure: the system does no more behave as required, i.e. it becomes impossible to satisfy the system's requirements.

Error: a perceivable system state that may lead to the failure (and precedes it), for example, the sensors can signal "low tire pressure". Recovery from this state may prevent more drastic consequences, e.g. a failure "impossibility to drive a car" because the tire is completely damaged.

Fault: the cause of the error, which will be discovered by some *diagnosis* (for example, "a hole in the tire").

A paramount example for a failure is, for a robot, the fact "battery empty" (i.e. "no power", and thus the inability to autonomously recharge itself). A possible, perceivable error that precedes this failure could be: "battery has a very high discharge rate" or "battery voltage under the threshold x". Corresponding faults would be e.g. "battery wear level too high", "a battery cell damaged", or "battery too old". However, the results achieved applying these definitions are not absolute, but highly relative to the designer's viewpoint and the perception abilities of a system. Supposing to have a (sub)requirement that states that a tire have to be flawless, the fault identified in the example above ("a hole in a tire"), would be the failure, while the error could be "a nail in the tire", which could be detected by inspection, before any loss of air pressure.

Faults are discovered by making a diagnosis, asking "Why did that error happen?" or, in other words, "What are the identified, known or hypothesised faults that cause the errors?". For the sake of simplicity, we decided to not capture *faults* explicitly in the model, since they are usually directly related to a recovery activity. Going on with the example, "a hole in a tire" will be recovered by an activity "repair tire", while a diagnosis resulting to a fault "tire valve open" would lead to the recovery activity "close tire valve". Another, more general, but less cost-effective repair activity would be to replace the tire with a new one.

The approach relates possible recovery activities directly to errors, skipping an explicit *diagnosis* phase for discovering the faults that cause an error. This is done following our aim to create intuitive and comprehensible models. *Relating recovery activities directly to errors, we capture the simplest form of diagnosis.* However, analogous to *Tropos* variability modelling, various recovery activities can be modelled for an error. *By defining pre- and context-conditions and contribution links for these recovery activities, the developer has the possibility to model "hints" for a diagnosis, which will* *consist in the selection of the best alternative recovery activity in a specific environmental context at run-time.*

More complex diagnosis mechanisms can be either added in the implementation phase or modelled as a first step of a recovery activity (e.g. introducing a diagnosis capability which selects alternative recovery activities depending on the result of the diagnosis. This diagnosis could possibly also rely on an expert system or learning mechanisms to find suitable recovery activities).

This approach does not give the same amount of flexibility at run-time as an exhaustive formalization of the system's goals, activities and possible states, together with planning algorithms. However, for complex systems situated in an unpredictable, non-discrete environment, a formal approach does not seem to be feasible.

Failure modelling is complementary to the modelling of goal failure conditions, which denote the impossibility to satisfy a goal and thus define its failure. In principle, an undesired state would also be modellable by a goal to avoid that state, but the idea is that following the "human way" of reasoning, it is often easier to enumerate undesirable situations than to precisely define desirable objectives. It is worth noticing that *Tropos4AS* failure modelling would, in a second step, also naturally fit to the idea of sending, at run-time, help requests to peers or dedicated (expert) systems which are able to provide new knowledge on possible errors and/or recovery activities.

3.2.4 Graphical modelling language

In this Section we briefly describe the graphical notation proposed to depict *Tropos4AS* models, which is founded on the *Tropos* modelling language, whereby the graphical representation for the new extensions was chosen to integrate with the existing one. The new concepts, depicted in Figure 3.7, are briefly described in the following.

Goal types (Figure 3.3) are graphically represented by a small circle put as annotation to the upper left corner of a goal, and containing the initial letter of the goal type. The inter-goal relationships *inhibition* and *sequence* can be represented graphically by arrows between goals, labelled «inhibits» or «seq», respectively.

The adopted graphical notation for the environment model is a simplified UML class diagram. Artifacts are represented as UML classes characterised by the functionalities they provide (as methods) and their state variables (as attributes), as in Figure 3.7 (b). They can be grouped into packages (e.g. to group artifacts external and internal to the system), and can be detailed, if necessary, by using UML class diagram relationships. Conditions can be modelled graphically by flag-shaped boxes linked to a goal and one or more artifacts in the environment. As illustrated in Figure 4.12, in the supporting tool, conditions and related artifacts are displayed in a table associated to each goal, defining the condition type, related environmental artifacts and a Java-like boolean expression evaluating the monitored values.

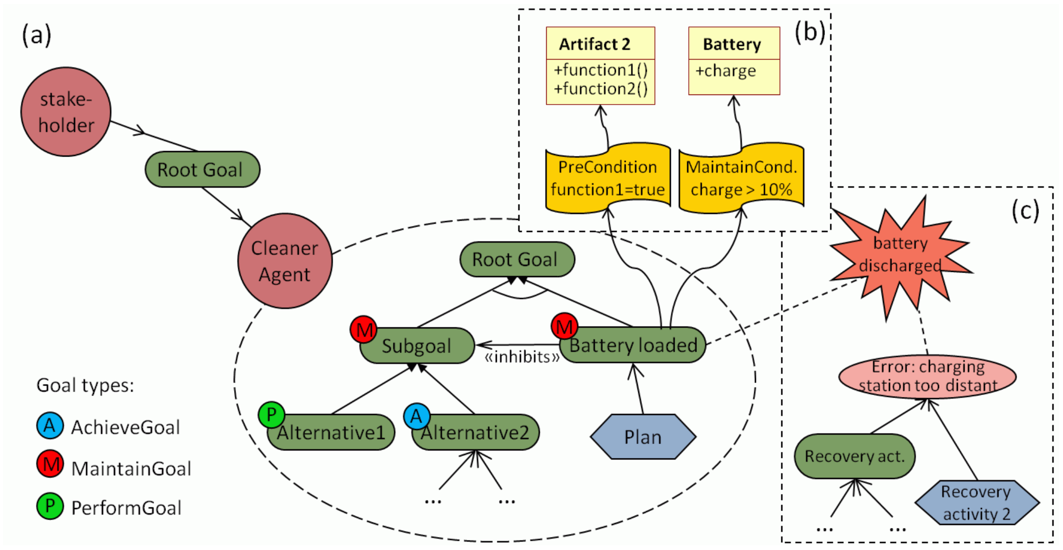


Figure 3.7: Graphical representation of the modelling concepts introduced in *Tropos4AS*. **(a)** extended goal model with goal types and new goal relationships; **(b)** environment model with conditions and artifacts; **(c)** failure model with failures, errors and recovery activities (which can also include parts of the original goal model)

.

Failures are represented by 'jagged' circles in the graphical notation (an example can be seen also in Figure 7.4) and are associated to the respective goal by a dashed line, while errors are represented by an ellipse. Recovery activities to mitigate errors are modelled by goals and plans by using the *Tropos4AS* goal model notation. Therefore, also parts of the original goal model can be reused.

These graphical representations are widely in line with an effort for unifying the graphical representation of popular goal modelling languages including O-MaSE and Prometheus [Padgham et al., 2008]. The chosen *Tropos*-like representation for goal models differs only slightly from ours, mainly by the symbols for actors and agents. The *Tropos4AS* concept of *condition* differs from the concepts of *trigger*, *message*, and *percept*, which are available in the proposed unified notation, and thus deserves an alternative symbol. Also the concepts of *failure* and *error* are not available, while for environment artifacts we propose to use the UML class-diagram notation, in line with the guidelines for the unification.

In the Taom4E modelling tool (described in Section 4.3), the modelling extensions such as conditions, goal types and new relationships are, represented as textual annotations and in tables. Graphical failure modelling is not yet implemented. However, errors and recovery activities can be captured with available goal, condition and means-end concepts, using a different colour key. A completely re-engineered version of Taom4E is currently under development, which bases on state-of-the-art GMF technology and will allow these extensions to the graphical modelling language.

3.3 Modelling Process

The extensions made to the *Tropos* modelling language in order to enhance modelling within the knowledge-level are able to capture an important part of the knowledge necessary for a self-adaptive system, directly in the system's goal model, i.e. the model of its requirements: dependencies to its environment, details on how high-level goals have to be achieved and on when they will be achieved.

Tropos4AS adopts the *Tropos* development process, as it is defined in [Penserini et al., 2007b], and extends its *Architectural Design* (AD) phase by adding the specific modelling activities ***Extended goal modelling*** and ***failure modelling***, for reacting to environmental changes and for preventing goal failure. It is complementary with respect to the previous *Tropos* AD, and collocated on the top of capability modelling. This aspect reflects the idea that *Tropos4AS* aims at supporting the design and implementation of the agent's knowledge level, namely the decision-making process behind the selection of capabilities (i.e. of a specific behaviour), as a paramount feature to cope with the development of self-adaptive systems.

Outline After a brief recapitulation of the *Tropos* modelling process, which helps to collocate the new modelling activities correctly, extended goal modelling and failure modelling are described. Afterwards, we present a simple pattern for the efficient modelling of variability in the requirements, which adopts the extended goal modelling.

Brief overview of the Tropos modelling process The original *Tropos* modelling process [Bresciani et al., 2004a] has four modelling phases. Our reference process [Penserini et al., 2007b] details this process into thirteen activities and enhances it, giving details on the modelling of capabilities. We focus on goal modelling, supported by the Taom4E tool, which respects the metamodels defined in [Susi et al., 2005].

In the Early Requirements (ER) phase, the stakeholders, their desires, needs and preferences, are captured and modelled in *Tropos* in terms of actors with their goals, softgoals, and dependencies.

The Late Requirements (LR) phase introduces the system-to-be as a new actor, to which stakeholders' goals are delegated. In the system goal model, these goals are analysed, applying a central activity of *Tropos*, the *goal modelling* procedure (reported in Figure 3.8), decomposing them to get more concrete sub-goals and design alternatives, and providing plans that will be means for their achievement.

```
**procedure** $goal\_modelling(g)$
$decision = decision\_on\_goal(g)$;
**case** decision:
 - *delegate* $g$ to existing actors or new actors $\{a_i\}$,
     $model.add(\{a_i\})$,
     $model.add(dependency(a_{depender}, g, a_{dependee}))$;
 - *expand* $g$ in subgoals $\{g_i\}$,
     $model.add(\{g_i\})$,
     $model.add(decomposition_{and/or}(g, \{g_i\}))$;
 - *contribute* $g/sg$ to goals/softgoals $\{g_i, sg_i\}$,
     $model.add(contribution(g/sg, \{g_i/sg_i\}, metric))$;
 - *solve* associating a plan $p$ to $g$, $model.add(means\_ends(g, p))$,
         *capability identification* for $g$, *capability_modelling(g)*;
 - *unsolvable* explore alternative goals;
```

Figure 3.8: The goal modelling procedure in the *Tropos* design process, as defined by Penserini et al. [Penserini et al., 2007b].

In the Architectural Design (AD) phase, a multi-agent system architecture is defined for the system-to-be, delegating goals and activities which characterise different roles, to different sub-actors. This step realises the fundamental software engineering principle of modularizing a software system into a set of subsystems that have high cohesion and low coupling. Architectural styles, such as the ones defined in [Castro et al., 2002] and [Penserini et al., 2006a] can be applied. The goals delegated to the sub-actors will again have to be analysed applying the *goal modelling* procedure, decomposing, delegating, and/or operationalising them. The obtained sub-systems are autonomous entities, that typically have various dependencies to other actors, with whose they are in charge of realising the objectives of the system-to-be.

In Detailed Design (DD), the plans defined earlier (representing the actor's capabilities, i.e. the concrete functionalities to be implemented) are detailed in the goal model and further by UML 2.0 activity and sequence diagrams.

Collocation of the *Tropos4AS* modelling extensions A goal model affords a first architectural decomposition of a self-adaptive system into components [Zhu et al., 2008]. We collocate our extended goal modelling and failure modelling for self-adaptive system in the *Tropos* AD phase, after the decomposition to sub-actors and the delegation of goals to them (specifically, step 7 in [Penserini et al., 2007b]).

Applying the *goal-modelling procedure* including algorithms to delegate, expand, contribute and solve goals (as in the algorithm in Figure 3.8), each sub-actor has its own goal model subtree detailing the goals it has to achieve. Each sub-actor represents a software agent that will be obtained applying the automated mapping which is part of the *Tropos4AS* framework. This detail is of particular importance if the engineer intends to implement the agents which reflect the goal model, using the proposed framework and tools.

A system could be implemented by a single agent for the entire goal model, or by one agent for each leaf node in the goal model. According to Zhu et al. [Zhu et al., 2008], a combination of these two extreme mappings, i.e. realizing a subtree of a goal model by an agent, seems more appropriate for many applications. [Zhu et al., 2008] also discusses problems arising in managing the sub-actors to satisfy the complete goal model. This problem is yet also appearing here, and can where it can be addressed by a centralised approach where a single actor delegates goal model subtrees to others, by applying architectural patterns as e.g. in [Castro et al., 2002] and [Penserini et al., 2006a], or by realising a collaborative, decentralised organisation as presented in Chapter 6.

However, if the focus of the software engineer is to capture the requirements of a system in its details, without going towards an agent implementation, the goal models in output of the *Tropos Late Requirements Analysis (LR)* phase (specifically, step 5 in [Penserini et al., 2007b]) could directly be used. A prototype representing the goal model of the whole system can also be generated from this starting point.

From here on, we will focus on the development of a single actor or "agent", and thus refer to this actor, which is itself defined by a goal model, as the *system* to develop. In the case reported above, or if the system is very small and is thus not decomposed, this *system* coincides with the system-to-be obtained from the *Tropos* LR phase.

The newly introduced modelling steps, together with the models in input and output of each step, are shown in the diagram in Figure 3.9. These steps will be performed iteratively and possibly also in parallel, though for clarity reasons only general feedback loops are depicted in the diagram. The modelling steps are explained referring to the CleanerSystem example, presented in Section 7.1.

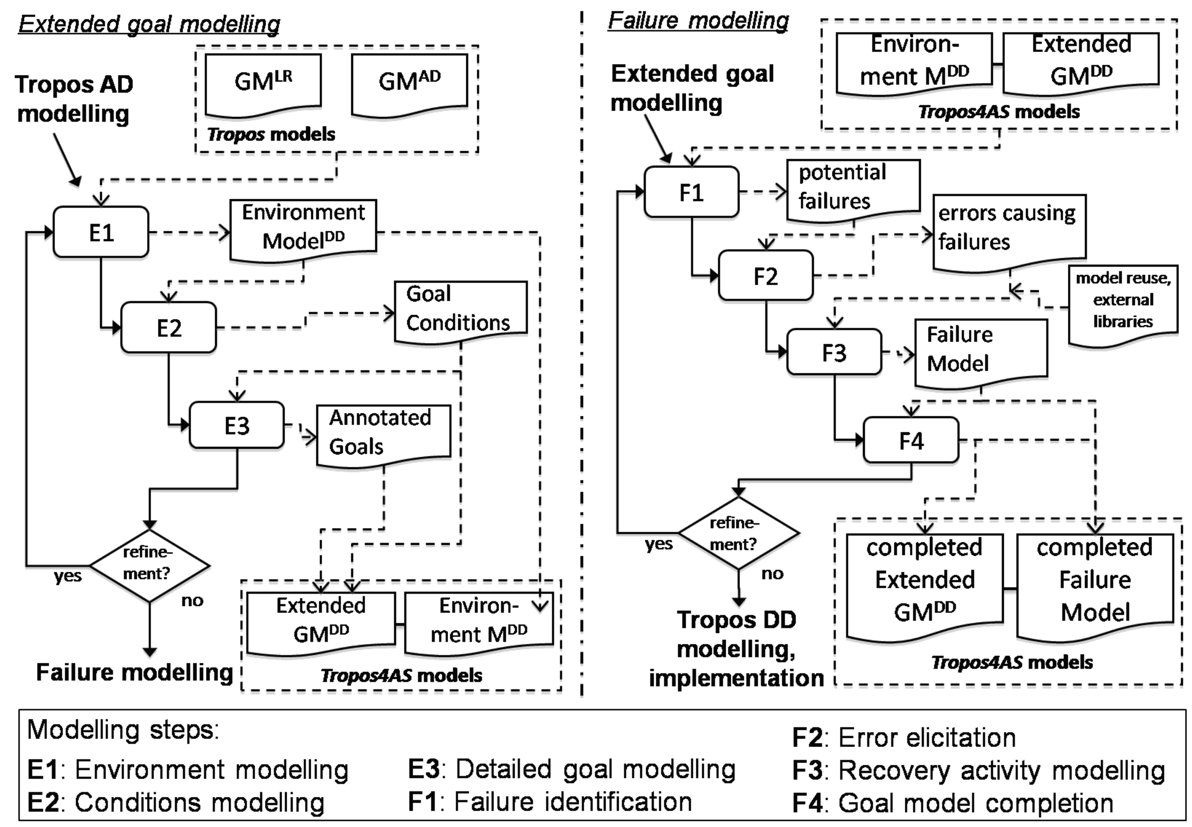


Figure 3.9: Steps of the extended goal modelling and failure modelling processes of *Tropos4AS*, including models in input and output of each step.

3.3.1 Extended goal modelling

The extended goal modelling of *Tropos4AS* includes modelling of the concepts introduced in Section 3.2.2: the surrounding environment, various goal properties like goal types and inter-goal relationships, and conditions concerning the goal satisfaction process. The process steps, depicted in the left part of Figure 3.9, are detailed in the following.

E1: Environment modelling

With the goal models resulting from the *Tropos* Requirements Analysis (GM^{LR}) and Architectural Design (GM^{AD}) steps (as defined in [Penserini et al., 2007b]) in input, the environment is modelled, eliciting the artifacts involved in the system and analysing the relationships between an agent's goals and these artifacts.

Artifacts are objects or tools allocated in the environment, either in- or outside the boundary of the system to develop (cf. Section 3.2.1). Agents can use and share them, to achieve their objectives. Please note that artifacts do not enclose *intentional* (or autonomous) entities in the system and in its organisational context. These were already identified and modelled as actors of the system-to-be in the earlier phases of the *Tropos* process [Bresciani et al., 2004a].

To identify the *artifacts* in- and outside the system-to-be, to whom the agent has a dependency to achieve a goal or to evaluate its achievement, the software engineer can ask modelling questions such as "which entities affect the achievement of a goal and can be perceived by the agent?" or "which (internal) artifacts does the system need to perform the requested plans?". Primary candidates to be represented as artifacts are the *Tropos resources* modelled in LR and AD goal diagrams, if they define objects used by the system to sense and/or act in the world (e.g. *broom* and *dirtSensor* in Figure 7.2). Artifacts can also be identified by eliciting the objects, services or tools inside or outside the system boundary, which an agent has to be acquainted with.

Additional artifacts involved in the system may be identified when modelling goal conditions in step **E2**. They can be continuously added to the environment model, iteratively performing the present step.

If available, domain models and ontologies can be used, either to identify artifacts or (on an instance level) to represent the model itself. For this purpose, the representation should be limited to a view of the relevant objects.

E2: Conditions modelling

This step captures the dynamic nature of the state of affairs associated to each agent's goals, by linking them to the domain entities relevant to the definition of this state. Operatively, we model the conditions that have to be fulfilled if a goal is considered to be achieved, as well as the conditions to be fulfilled for the creation (activation) of a goal.

Conditions are captured by relating goals to the corresponding artifacts and operationalised by boolean expressions, to evaluate monitored values, accessing data and invoking sensing functionalities offered by these artifacts. Furthermore, conditions can also depend on the state of another goal or on an interaction with other actors in or outside the system-to-be. An example can be seen in Figure 3.7, where an artifact representing the device's battery is related to the corresponding goal for being able to monitor the battery charge level. We recommend to write the boolean expressions using a JAVA-like syntax, in the form

$$\mathsf{artifact.functionality(parameters)}\ [=,<,>,\ldots]\ \mathsf{expression}$$

to have a common denominator and to reduce the effort of a mapping to the implementation. For high-level models, which will not necessarily mapped to agent code, also natural language can be used.

The reason for an agent to decide to bring about that goal at a given instant (i.e. the adoption of a goal) is made explicit by a *creation condition*. To restrict the adoption of a goal to a particular context, *preconditions* can be modelled. To define the achievement of a goal, an one-time *achievement-condition* or a *maintain-condition* (for the goal to maintain some state in the long term) can be defined.

Failure conditions denote states from whose it would be impossible to achieve a (sub-)goal, e.g. cleaning tool worn out for the goal OfficesClean or battery broken for the goal MaintainBatteryLoaded. The satisfaction of such conditions denotes the failure of the goal. While failure of a root goal is critical and will lead to system failure or at best to a system working in a degraded mode [Avizienis et al., 2004], failure of sub-goals can be caught at a parent goal level, e.g. by satisfying alternative subgoals.

A state in which it is no more necessary to try goal satisfaction (especially for maintain goals) can be expressed by a *drop condition*, whereas a state in which goal achievement should be temporarily suspended can be modelled by a *context-condition*.

Making these conditions explicit, new monitoring requirements may be identified, leading to new artifacts or new sensing functionalities in the environmental artifacts. The environment model should revised in parallel (step **E1**), adding missing artifacts and new functionalities.

E3: Detailed goal modelling

In **E2**, by modelling conditions, the goal model was enriched with important details on the achievement of system requirements. In this step, further details are given to the goal model, in order to obtain details on the goal satisfaction process at run-time. By analysing the conditions specified in **E2**, the type of each goal is inferred: *maintain*, *achieve*, or *perform*. For example, maintain and achievement conditions directly characterise the respective goal type. Figure 3.3 shows the possible conditions that characterise each type of goal. Goal types technically detail the underlying goal satisfaction process and the role of conditions in this process (cf. [van Riemsdijk et al., 2008, Morandini et al., 2009b]).

Note that introducing goal types in a goal model, without any criteria, can lead to a hardly predictable system behaviour, a characteristic vividly undesired for a design framework. A main design guideline to reduce the possibility to have unpredictable system behaviours is that in each path from the root to the leaf goals there has to exist at most one maintain-goal.

To define further constraints on the possible goal satisfaction workflow, inter-goal relationships can be made explicit:

Goal sequence is often desired when composing a goal into subgoals, which have to be achieved one after another, rather than in parallel.

Inhibition between goals can be necessary if two goals are in conflict. Rather than letting the agent decide (i.e. by its goal deliberation mechanism), the modeller can set such a precedence relationship to define goals that are more critical than others. This will be useful with maintain-goals, which are active for a long time period. For example, a goal MaintainBatteryLoaded should inhibit MaintainPlaceClean.

.

After defining goal types and goal relationships, by iterating on the whole process (feedback loop in Figure 3.9), the goal conditions in step E2 can be reworked and completed to comply with the chosen goal types, respecting the metamodel (Figure 3.3), and having modelled and detailed all relevant environmental artifacts used in conditions. For example, an achieve-goal may have a creation and an achievement condition, whereas maintain-goals can have maintain- and target-conditions, to start and to stop goal achievement, respectively.

However, we want to note that the purpose of the steps E2 and E3 is not to define goal models formally and completely, but to capture and specify the requirements and the known information about what an agent has to observe and consider for goal achievement.

3.3.2 Failure modelling

With extended goal modelling, the *Tropos4AS* process allows to model details of the nominal behaviour of a system. The objective of failure modelling is now to provide a process for completing the elicitation of the requirements of a system, by capturing also its exceptional behaviour. The process is summarised in the right part of Figure 3.9 and provides support for designers to model potential failures in goal achievement, capture unwanted states of affair (errors) that could cause these failures, and identify recovery activities to be performed either to anticipate foreseeable errors or to recover from unpredictable ones, to avoid goal failure

The reaction to exceptional situations (i.e. the recovery activities) may finally be modelled either within the core part of a goal model, e.g. as an alternative plan or goal tree, or kept aside, according to the principle of separation of concerns. This choice is however highly dependent on the current application domain and on the designer's preferences. For example, it may be the case of seldom events that bring about a deviation from an agent's nominal behaviour that would be, hence, preferably modelled externally to the core part of the goal model which, on the contrary, describes its nominal behaviour. While, it may be the case that recurrent events recognised to be potential causes for failure would lead to a modelling within the agent's goal model, in the form of goals to achieve or maintain.

F1: Failure identification

The designer analyses each goal of an actor in the system to develop, for the possibility of failures in the goal achievement process. Failure denotes a situation characterised by a set of states (of the world and of its own), in which an agent (with its actual knowledge and with the given capabilities) is unable to reach a goal.

Failures can be identified analysing the scope of a goal, the modelled conditions, and the (often unexpected) variability in the environment of the system. Identified failures (e.g. a failure unable to clean for a goal MaintainPlaceClean) are explicitly captured graphically and connected to the corresponding goal, that is, to the requirement that could possibly not be satisfied.

First candidates for associating a possible failure are goals with failure-conditions or drop-conditions, made explicit in step **E2**, in the case that the resulting goal failure is not handled at a higher level (e.g. by modelled alternatives to achieve a parent goal). Vice-versa, failure identification may also lead to a revision of goal failure and achievement conditions (in step F4).

In general, an analysis of failures may be of particular importance for goals that have yet no alternative, e.g. goals in an AND-decomposition and, in particular, root goals, that are, goals directly delegated by the stakeholders, without any parent goal that could mitigate the failure. Failures can be of various kinds:

(a) *predictable by monitoring* for particular situations, which we define as *errors* (e.g. the value for dirtiness after cleaning exceeds some threshold). These failures can be anticipated by various recovery activities, depending on the domain and the designer's choices:

(1) at design time by implementing missing capabilities (e.g. for a cleaner robot, the inability to clean from a specific type of dirt).

(2) at design time by modifying the decision criteria for selecting some behaviour, changing conditions, contributions and goal relationships.

(3) at run-time by using alternative capabilities.

(4) at run-time by using a novel combination of existing capabilities (e.g. a broken tool could be substituted by a proper use of other available tools).

(b) *not predictable by monitoring* for errors, i.e. the only perceivable errors cannot be mitigated avoiding to lead to failure, and are thus already states of failure (e.g. sudden battery failure or a crash of the robot). Such failures can be mitigated at design time by:

(1) making the system more robust to the problems (e.g. by implementing a redundant power supply), or by

(2) giving the ability to detect (some of the) problematic situations (errors) that lead to the failure (e.g. by employing special sensors for distance measurement, to notify an imminent failure before it is too late to react). In this case, the failure becomes perceivable.

A brief description of the failure in natural language can help to conduct the next steps.

F2: Error elicitation

Now, to anticipate these failures, the designer tries to discover the *errors* leading to them – perceivable states, in which the system will become unable to achieve its goals.

Following the modelling question “What could be the cause for such a failure?”, errors are forecast at design time, according to a top-down process. Similarly, errors and correlated failures can also be found in a bottom-up process, analysing software and hardware problems that can happen outside the agent boundary and affect the agent’s work. In this case, the design questions are “Are there requirements that could no more be satisfied (completely) if that problem occurs?”, or, in other words, “What goal failure could be caused by that problem?”, and, to identify the error for which the system has to be monitored, “How can the problem be perceived?”.

Errors are first defined in natural language (e.g. dirt not removable or cleaning tool broken) and, related to the environmental artifacts concerned for monitoring the particular system state (e.g. dirt sensor) or representing the physical entities (e.g. cleaning tool).

Leading, by definition, directly to goal failure, goal failure conditions modelled in E2 describe an error. It however depends on the intention of the designer, to decide if he modelled conditions on purpose to let the goal fail (e.g. because this failure would be handled on a higher level) or if he should proceed with error modelling to try to prevent failure.

Errors can be categorised into errors from which recovery is possible before they lead to a failure, and errors from which timely recovery is impossible.

For each error identified, it can be decided to not mitigate it because it involves only a reduced risk, or because mitigation would be more expensive or more time intensive than human action necessary in the case of system failure. To evaluate risks in detail, a separate risk analysis (e.g. as proposed in [Asnar et al., 2006]) can be performed.

F3: Recovery activity modelling

This step consists in modelling possible recovery activities that can be executed to recover from an error, or in alternative, and depending on the domain, to prevent it. Recovery activities are modelled by *Tropos* concepts, and can contain single plans or pieces of a goal model. Thanks to the *Tropos* conceptual notation, in this phase the designer can abstract from the complexity of techniques related to error diagnosis, creating intuitive and simple models that relate recovery activities to errors.

Modelling of recovery activities depends on the kind of failure along with its severity, and on the possibility to recover from errors to anticipate failure. Possible approaches are illustrated in the following:

- anticipation of goal failure (on failure of its subgoals): the recovery activity can consist in a new capability or a new subgoal executed in alternative, when the specific error conditions are present.

- anticipation of failure of available capabilities (or of entire subgoals) caused by an undesired environmental state: the recovery activity should be executed at the time the error is detected, to recover from it before executing the capabilities, which otherwise would possibly fail.

- failures of type *b*: in this case, the best we can do is to try to reduce the probability of system failure, e.g. by making the system hardware more reliable (for example, by mounting a backup battery to prevent power failure), by employing new sensors, or by restricting the operation environment.

To analyse and detail each of the goal model fractions modelled as recovery activities, in this step, the plan modelling process (Step 10 in [Penserini et al., 2007b]) can be applied, possibly also reusing parts of the existing goal model. For each of the new goal model parts it can also be necessary, depending on the size of the part and the desired level of detail, to perform the whole extended goal modelling process from steps E1 to F3. For example, recovery activities will often have to be bound to a specific context, by defining pre-conditions (thus applying a simple form of diagnosis). Moreover, modelling positive contribution to non-functional requirements can be a further means for selecting between alternative recovery activities.

F4: Goal model completion

Finally, the designer has to decide if the recovery activities defined in the previous step should be part of the normal workflow of the system (and thus becomes part of the goal model) or if they should remain a representation of an exceptional situation, which would not be part of the agent's nominal behaviour.

If the execution of the recovery activity depends on a specific environmental context or situation that will happen in a normal workflow not considered exceptional, the piece of goal model representing the recovery activity can be integrated to the original goal model of the agent, as an alternative capability that can be used for achieving the related goal. In this way, variability is added to the goal model, extending the agent's adaptivity also to situations that would previously have been leading to goal failure. Modelled errors can be integrated to the pre-conditions, if the execution of the recovery activity should be limited to this case. Goal modelling (Figure 3.8) and the extended goal modelling introduced within this chapter should be reiterated after this step, if non-trivial changes to the goal model occurred.

In the case that the error is considered to be a rare event and recovery from this error should not be part of the agent's main activities (e.g. a hardware or software failure or a problem in inter-agent communication), the recovery activity will not be integrated into the goal model, but, following the principle of separation of concerns, it will be kept within the failure model and remain separately, until run-time. In this case, we recommend to put some effort into defining the error, using a JAVA-like syntax, by a boolean expression related to the interested artifact, analogous to conditions (see step E2). From a technically viewpoint, with the mapping to run-time that is presented in

Chapter 4, eventually, at run-time, also the failure model will be part of the ordinary goal model. However, the agent will have no opportunity for achieving goals that are part of the failure model, despite in the case that one of the identified errors occurs. In this case, a creation condition relative to the error will become true, and thus one of the associated recovery activities, whose preconditions are actually satisfied, will be executed.

Being external to the goal model, at run-time the failure model can also be independently updated with new failures, errors and recovery activities. Despite it is outside the scope of this journal, employing a learning mechanism that keeps track of successful application of capabilities would also be feasible with this approach. This would contribute to the realisation of a feedback mechanism, where designers are informed of changes in the failure model and can consider them for design time integration in the subsequent development cycle.

3.3.3 A pattern for modelling variability

Taking advantage of alternatives modelling together with an optimal use of the extensions introduced for modelling self-adaptivity, we provide a guide to model *alternative configurations* (high-level alternatives) in a goal model, together with the decision criteria to select between these alternatives, enabling a simple form of self-adaptation of the system to different contexts. The modelling pattern, illustrated through a simple cleaner robot example, considers the goal model of a single system and conducted with the aim of supporting a dynamic, optimized alternatives selection at run-time.

The following is the desired behaviour that the system should exhibit: initially, the optimal configuration for the actual environmental context is selected. If the top goal of the actual configuration fails or the context dramatically changes in a way that the system in this configuration is no more appropriate to reach its goal (i.e. some context conditions are negative), the system has to select an alternative configuration. If the top goal of the current configuration is actually achieved, but at this point a configuration is available which would give more appropriate results in the current context, the system should select this new configuration and try to achieve it, too.

Proposed approach The structure of the pattern is sketched in Figure 3.10. Within goal modelling, special effort has to be made to identify high-level alternatives to satisfy the system's root goals. These high-level alternatives define the main behaviours (also *target systems*, e.g. in [Sawyer et al., 2007]) that will be implemented. They are

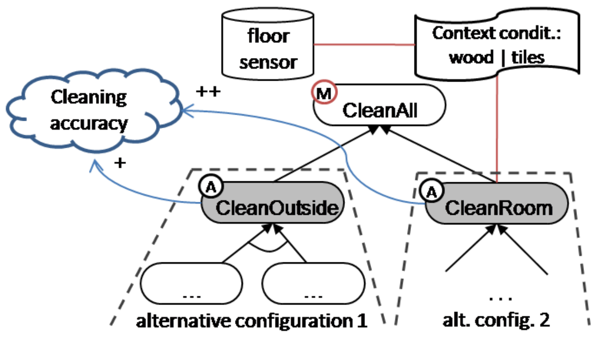


Figure 3.10: Structured modelling of high-level alternatives.

represented by OR-decomposing root goals, which are typically maintain-goals (e.g., CleanAll in Figure 3.10), to achieve-goals (e.g. CleanRoom and CleanOutside).

Subsequently, the obtained alternatives can be constrained by modelling context conditions (flag-shaped boxes in Figure 3.10). These conditions have to be true during the whole goal achievement process. Generally, unconstrained alternatives to the achievement of a goal give typically more general, but less optimized solutions than alternatives tailored to a specific context.

Moreover, alternatives can be characterised by modelling positive or negative contribution links to softgoals (represented as clouds), which represent the system's non-functional qualities. Softgoal contributions capture the preference of one alternative above others, regarding various quality aspects. Considering also a maximisation of contributions to softgoals as a decision criteria, the selection of alternatives can be optimized. Additionally, giving more or less importance to the various softgoals modelled, this selection can be influenced by the stakeholder or the user at run-time.

Application: behaviour To illustrate the application of the modelling pattern, we sketch the system behaviour that would be achieved implementing a cleaning robot called CleanerSystem, which contains the model fragment displayed in Figure 3.10, following the mapping to a BDI-based implementation defined in Chapter 4.

Suppose that hovering and mopping are the plans implemented for the achievement of the goal CleanRoom. Both of them achieve an accurate cleaning, but will fail when gravel is sensed. On the other hand, the plans implemented for the achievement of CleanOutside provide cleaning of coarse dirt, but are not able to clean properly from fine dust. We evaluate the behaviour of the system by looking at the expected execution of a scenario, where gravel is sensed in a room with tiles on the floor, while the agent is in configuration CleanRoom (which has a higher contribution to the softgoal Cleaning accuracy).

The agent configuration CleanRoom is not suitable for this type of dirt. The plans will fail and thus also the goal CleanRoom. The system tracks back to the parent goal CleanAll, adapts to the second possible configuration, with goal CleanOutside, and can now clean the floor from gravel. Once the floor is clean from gravel, the goal CleanOutside is achieved. Notice that, in the traditional *Tropos* OR-decomposition, achievement of CleanOutside would cause the achievement of the top level goal CleanAll. Thanks to the features of *Tropos4AS*, the maintain-condition of CleanAll is not yet satisfied, and now the sub-goal CleanRoom is again achievable, having no more gravel in the office room. CleanRoom will thus be reactivated to clean the remaining fine dust in the office. In another scenario, where the CleanerSystem has to clean the access road, because of the context condition modelled, the goal CleanRoom would never be selected, so the robot would only clean for gravel.

3.4 Final Considerations and Related Work

In this chapter we introduced central parts of *Tropos4AS*, a conceptual model to capture essential properties of self-adaptive software systems, and a development process to construct these models, collocated within the *Tropos* methodology. *Tropos4AS* focusses on knowledge level issues important for the decision making process at run-time. It inherits the ability of *Tropos* to capture variability in goal models at requirements analysis time and to deal with alternatives also during the system design phases. To give to a self-adaptive system the awareness about the environment it is embedded in, *Tropos4AS* allows to model the artifacts in the environment perceived by an actor in the system, and to capture the effects of changes in it to the goal achievement process.

One of the key features of an effective implementation of a self-adaptive system is an explicit representation of its requirements at run-time. Envisioning a run-time representation of goal models, such as the one proposed in the next chapter of this thesis, we enrich the models with detailed information on the goal satisfaction process, to detail goal achievement and alternatives selection dynamics. Various conditions on the context in which a goal would be achievable and on goal failure can be captured, and conflicts between goals can be expressed by basic relationships.

To be able to act properly in uncertain domains, *Tropos4AS* assists the designer in the analysis of possible requirements failures. Some degree of failure tolerance is achieved by identifying, at design time, possible failures, the errors causing them, and proper recovery activities. This failure modelling process helps to elicit missing functionalities, which can be integrated to the system goal model, or be kept separate, to differentiate the exceptional from the nominal behaviour of the system. Moreover, the failure model gives an interface for the application of domain-specific diagnosis techniques.

Together, these extensions contribute to a detailed representation of a system's requirements specification in a goal model, together with the decision criteria that define the adaptive behaviour to bring about these requirements in a variety of environmental contexts.

Various approaches extend goal models to capture variability in goals and the environment and details for goal satisfaction, aiming to the development of autonomous and self-adaptive systems [Lapouchnian et al., 2006, Liaskos et al., 2006, Ali et al., 2010]. However, in these works, goals lack of important run-time concepts such as a life cycle for goal activation and achievement, and no development process is defined. In [Ali et al., 2010], goal models are completed with annotations to bind the environmental context of an application to requirements alternatives, aiming at mobile applications. Our framework covers not only such pre-conditions, but also conditions that define goals in a declarative way (defining achievement, failure, etc.) leaving the final decision on how to achieve goals, to the software at run-time. On the other side, declarative specifications of goals with temporal logic, as in Formal Tropos [Fuxman et al., 2001], KAOS [Dardenne et al., 1993] and GTD [Simon et al., 2005], are amenable for formal verification, but are too restrictive for our purposes, since we aim to an execution by BDI languages, with a goal satisfaction behaviour as defined in Chapter 5.

The goal models proposed in [DeLoach and Miller, 2009] define positive and negative triggers for goals. These relations are comparable to *Tropos4AS* creation and drop conditions, but work on an instance level. Detailed restrictions for the use of these additional relations are formalised. This can be considered as future work for *Tropos4AS*.

[Goldsby et al., 2008] and [Berry et al., 2005] define self-adaptive system with goal models, using *i** and KAOS, respectively. They propose goal-oriented modelling of every possible system configuration in a distinct goal model, and also of the possible transitions between configurations. The main difference with respect to our approach is that we avoid to specify every system and every possible adaptation between systems, which may shortly lead to a bottleneck and to numerous, mostly redundant models. Instead, we model a single system that includes all high level variability. Moreover, our approach does not specify transitions between alternatives, but defines the opportunity for the system to select an alternative behaviour, in a specific environmental context. The arising problem of managing complex models can be tackled e.g. by displaying different views of a model.

In the goal-based modelling approach recently presented by Cheng et al. [Cheng et al., 2009b], similar to *Tropos4AS* failure modelling, environmental conditions that pose uncertainty at development time are identified, to uncover places in the model where the requirements need to be updated. Failure is mitigated by adding low-level subgoals, by making existing requirements more flexible using the RELAX language [Whittle et al., 2009], or by adding new high-level goals to define a new target system to which to adapt. Also [Baresi and Pasquale, 2010] follow an approach similar to ours, taking inspiration from KAOS *obstacle analysis*. The approach models *adaptive goals*, which identify conditions for goal violation, similar to our error conditions, and define countermeasures for recovery. The work aims at an instantiation on a service platform. Also aiming at service-based applications, [Qureshi and Perini, 2009] links goals to an ontological representation of the surrounding environment, to capture alternatives together with their monitoring criteria. Since *Tropos4AS* noes not stick to any particular language for giving a detailed model of the environment, it can be integrated with such approaches.

The *Tropos4AS* models capture the knowledge level of a system, i.e. a high level description of the system's goals and behaviours, but they do not detail the implementation of the single capabilities (i.e. *Tropos* plans). Moreover, the approach makes no assumption on specific reasoning methods and implementation architectures beyond the availability of basic BDI concepts. The scalability of the modelling approach remains an issue, which is addressed by focussing on the single sub-actors of the system, while stability of adaptations can be achieved by modelling proper goal conditions.

The effectiveness and comprehensibility *Tropos4AS* models, in comparison with *Tropos* models, was evaluated in an empirical study with subjects (see Chapter 8), which gave encouraging, positive results.

Chapter 4

From the Model to Run-Time

After having introduced, in Chapter 3, various extensions to the *Tropos* modelling language for capturing the inherent properties of self-adaptive systems, together with the modelling steps for their use, now we collocate these extensions in the big picture of the *Tropos* development process and carry this process on to an implementation and run-time phase.

A mapping of concepts and relationships from *Tropos4AS* goal models to a BDI-agent architecture is then presented, which preserves the representation of the requirements in form of a goal model at run-time. This model can be navigated for monitoring and decision making, and eases traceability of run-time artefacts and decisions back to the design and requirements phases. As target language and execution platform, we adopt the goal-directed BDI agent framework Jadex [Pokahr et al., 2005], taking advantage of its explicit, persistent representation of goals at run-time.

In the third part of this chapter we present the ***t2x*** (*Tropos4AS to Jadex*) mapping plug-in, and extensions to the TAOM4E modelling tool, developed to support the modelling and mapping process for the development of software agents with *Tropos4AS*. Automated code generation results to an executable implementation of an agent prototype, which exhibits a basic goal-directed behaviour with failure handling and adaptation to the environment, exploiting modelled variability and softgoal contributions. The complete tool-supported modelling process is finally illustrated.

4.1 The Process: Overview

We collocated *Tropos4AS* extended goal modelling and failure modelling in the *Tropos* Architectural design phase, giving the focus on the knowledge level of the system's sub-actor goal models (refer to Section 3.3 for details). The process artefacts are displayed, starting from *Tropos* goal models obtained in the Late Requirements Analysis phase, in Figure 4.1.

The (automated) mapping, introduced in the present chapter, is applied to these extended models, to create a representation of the goal model at run-time, which is interpretable by the Jadex agent platform. This mapping reduces the conceptual gap between agent design and run-time and provides an immediate implementation of prototypes which reflect the high-level behaviour represented in the goal model. A middleware layer implemented in Java ensures an implementation according to the intended goal model semantics, where it is needed to go beyond the possibilities given by the declarative BDI language provided by Jadex.

For the environment, we further propose a mapping to UML class diagrams, and the use of available UML tools for editing and implementing them. This step can be appropriate for the creation of prototypes and simulations, while in various other cases an API for access to the environment would be already available. We also made some experience with the modelling of the environment by an ontology (using an editor such as Protégé) and a generation of access interfaces from this ontology, with the Jadex *Beanizer* tool. To run an agent instance which implements the goal model, its definition can be loaded and executed on the Jadex platform.

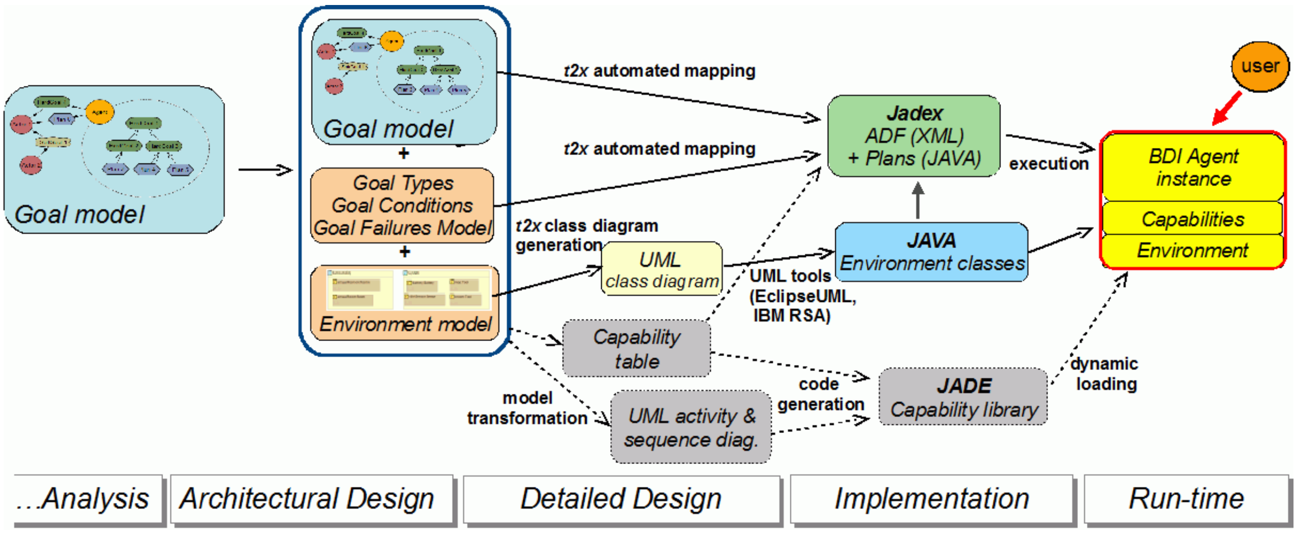


Figure 4.1: Overview: collocation of the various tool-supported Tropos4AS development activities and design artefacts, going towards a BDI agent implementation. Dotted lines: Complementary capability implementation approach, presented in [Penserini et al., 2006b].

The detailed design and implementation of the *capability level* (cf. page 35) is not part of this thesis. For completeness, in Figure 4.1, the capability modelling and implementation approach by Penserini et al. [Penserini et al., 2006b] is depicted by dotted lines. It includes the detailed modelling of capabilities (i.e. *Tropos* plans in *means-end* relationship to goals, together with their eventual plan decomposition) by UML activity and sequence diagrams, whose skeleton is created by model transformation. The approach also includes a tool for code generation to *JADE*[1] *behaviours.* We envision an implementation where capabilities are stored in a pool and dynamically loaded at run-time when needed by an agent instance.

4.2 A Mapping from Goal Models to BDI Agents

Although in the requirements analysis and design phases we did not commit to a specific implementation framework, language and architecture, for the implementation we refer to a BDI-based architecture, which provides an explicit notion of goals and plans – notions extensively used in the *Tropos4AS* models.

In *Tropos4AS* we aim at supporting the knowledge and reasoning aspects of adaptive and autonomous agents. The BDI model fits very well with our scope of having a clear separation between knowledge abstractions (i.e. goals and beliefs) and concrete actuators (i.e. plans) to sense and affect the environment. Moreover, BDI-based architectures provide belief monitoring and a deliberation mechanism for goal selection.

In the following, we provide a mapping process from a subset of Tropos4AS concepts and structures to a programming language for BDI agents, with a dual aim: first, to have an explicit representation of the goal model, navigable at run-time, and second, to obtain an interpretation of this goal model at run-time that follows the intended semantics[2] *of the model and respects modelled decision criteria.*

Concretely, we adopt the Jadex Agent Definition language. Jadex (an overview is given in Section 2.4.2) was selected for its explicit representation of goals, which allows to have a homogeneous mapping between concepts available at design time

[1]JADE (Java Agent Development Framework) [Bellifemine et al., 2007] is an agent platform which provides FIPA-standard messaging functionalities and the execution of agent *behaviours* in independent threads.

[2]The intended semantics of *Tropos* modelling constructs were defined informally in [Bresciani et al., 2004a] and detailed for various purposes in several works (e.g. [Fuxman et al., 2004, Penserini et al., 2007b]. Details of our interpretation were further elicited by analysis of existing models and discussion with researchers that use the methodology.

and at run-time, and for the use of the industry-standard languages JAVA and XML. Nevertheless, the *Tropos4AS* approach remains mostly technology-independent and the mapping would be applicable similarly to other BDI-based agent platforms.

Targets of this mapping are the development of agent-based systems with adaptivity properties as well as a rapid construction of prototypes of an executable behavioural model of the modelled system.

4.2.1 The considered subset of *Tropos4AS* concepts and structures

The *Tropos* metamodel (a relevant part is shown in Figure 2.3), and thus also *Tropos4AS*, allow a multitude of different relationships between the modelling concepts of actor, goal, plan and resource.

An attempt to limit the relationships usable in *Tropos* models was proposed by Estrada et al. [Estrada et al., 2006]. This approach goes top-down, removing from the metamodel the relations that are redundant or that have no meaningful semantics. For our purpose, we use a pragmatic bottom-up approach, to limit to a small, but not necessarily minimal set of possible relationships, which is usable and intuitive and has clearly defined semantics. Most of these limitations are also present in the modelling tool Taom4E, thus limiting already the creation of goal models. Since we focus prevalently on the knowledge level, for conciseness, we do not consider *Tropos* resources. The considered subset contains the following:

- The basic constructs of actor (here, an actor in the system to implement, i.e. an *agent*), goal, softgoal, and plan.
- A limited set of relationships between these constructs:

  AND/OR-decomposition A homogeneous[3], acyclic relationship to decompose goals, softgoals, and plans hierarchically into sub-entities.

  Means-end-relation A relationship between a plan (the means) and a goal (the end). Relations between softgoals and goals, with the same semantics, can be modelled more accurate with contributions, while homogeneous relationship should be modelled by OR-decompositions.

[3]homogeneous: a relation restricted to entities of the same type, e.g. goal – goal.

Dependency A relationship between two actors, with a goal, resource or plan as dependum and a goal or plan as the cause of the dependency (the *'why'*-argument).

Contribution We limit to contribution relationships from a plan or goal to a softgoal.

Like in our whole work, if we speak about *goals*, we always refer to *hardgoals* (as opposed to softgoals), unless specified differently. Goals include *Tropos4AS* extensions for types, intra-goal relationships and conditions. Softgoals include *importance* as a property which should be changeable at run-time. Moreover, we include *Tropos4AS* concepts regarding the environment and failure models.

All relationships are non-reflexive (i.e. an entity cannot be related to itself) and can be n:n (e.g. a goal can be composed to n subgoals and, vice versa, subgoals can belong to more than one decomposition, having n parent goals). The semantics for each relationship are recalled while explaining the mapping. Regarding the AND/OR-Decomposition, coming to *Tropos4AS* goal types and conditions, these semantics become more intricate. They are formalized in Chapter 5.

4.2.2 BDI concepts and mapping guidelines

The target BDI architecture used in this mapping is built around the notion of *agent* endowed with *goals*, *plans* and *beliefs*, concepts identified already in [Rao and Georgeff, 1995] for a practical implementation of BDI agents, and are now implemented in similar fashion by various goal-directed, event-based agent frameworks such as Jason [Bordini and Hübner, 2005], Jack [Winikoff, 2005], and Jadex. In the following, we report some aspects essential for the mapping, which uses the terminology and semantics of the Jadex Agent Definition language.

In brief, a *goal* denotes in general some target to be reached, a *plan* denotes a list of activities that can be performed, while the *belief base* holds various facts and beliefs about the system and the world. The *agent* is the execution unit, with messaging functionalities and an independent thread of control.

Figure 4.2 shows the available mechanisms for activating goals and executing plans. Goals can be activated (or *"dispatched"*) by creation conditions evaluated on the belief base, from inside a plan, or they can be active from beginning (cf. *Formal Tropos* model simulation, where all goals are interpreted in this way). Active goals can be pursued by executing plans; that is, they act as *triggering events* for plans which were

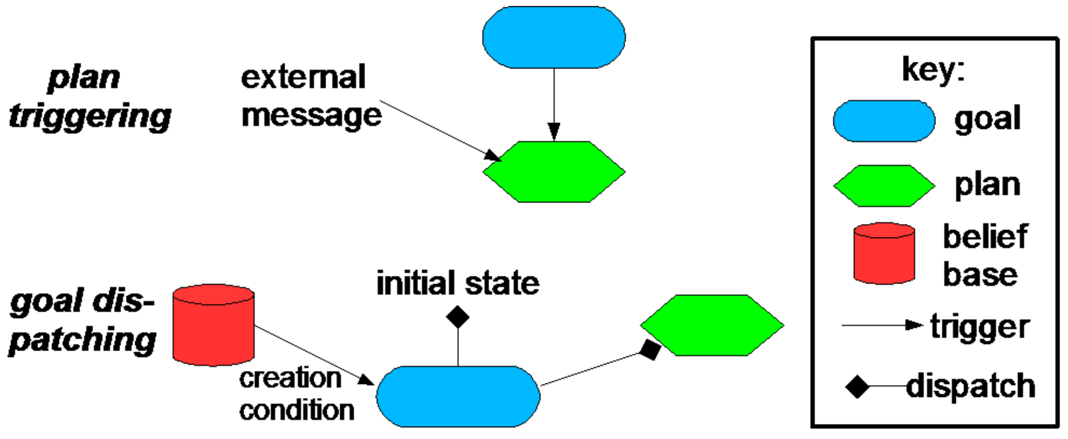


Figure 4.2: Goal activation and plan execution mechanisms in the target BDI model.

defined to be a *means* for their fulfilment. Moreover, plans can be executed on request by an external message (e.g. to handle another agent's requests).

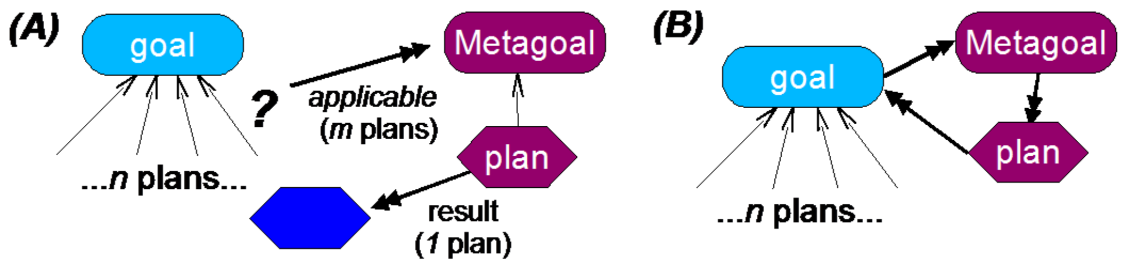


Figure 4.3: The Jadex meta-reasoning mechanism: **(A)** representation of the selection process applied when more than one plans are applicable in the actual context; **(B)** graphical representation of the meta-reasoning concept, as used in the following Figures 4.6 and 4.7.

Modelling of alternatives and their decision criteria is a main concern of *Tropos4AS*. At run-time, having available various plans that could be executed to pursue a goal, a selection has to be made. Jadex provides the concept of *meta-reasoning*, a mechanism for the selection between plans, each time two or more alternative plans are available as means to satisfy a goal (Figure 4.3 **(A)**).

Each time a selection has to be made, on purpose for each goal meta-reasoning is activated and selects one plan, either respecting some predefined sequence (default behaviour), or applying user-defined selection mechanisms. The various types of goals and conditions will be discussed within the mapping of *Tropos4AS* concepts.

The specification for the mapping has been conducted along two lines: mapping of *Tropos* concepts and structures, and mapping of the *Tropos4AS* extensions. The following guidelines were observed:

1. Map concepts and relationships available in the BDI model, one-to-one.
2. Map concepts and relationships that can be expressed in the BDI model by a comprehensible and clearly delimited combination of concepts, one-to-n.
3. Map relationships to a definition by imperative language concepts, if the declarative BDI language concepts do not suffice for an efficient mapping.

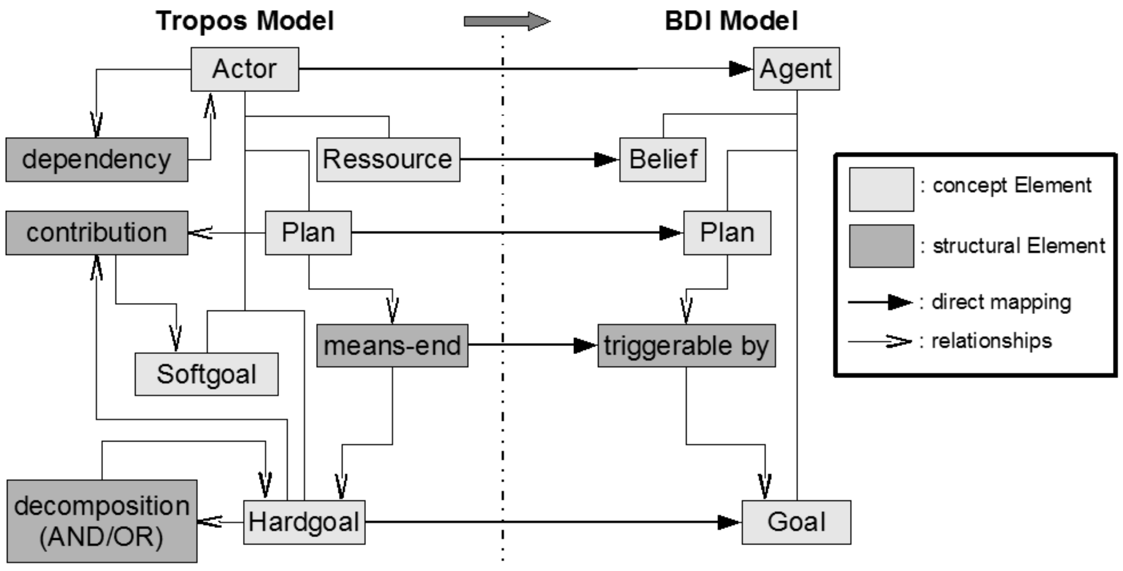


Figure 4.4: Tropos concepts which have a direct mapping to the adopted BDI architecture.

4.2.3 Mapping of *Tropos* concepts and structures

Hereafter we describe the mapping of the various *Tropos* concepts and structures to the BDI architecture identified in the previous section. The need of this mapping is emphasized mainly by the fact that **Jadex, analogous to other existing goal-oriented agent programming languages, such as Jason and 2APL, does not support goal hierarchies as a native feature, but supports the creation or activation of new goals from plans.** Actors, (hard)goals, plans and resources can be directly mapped to the relative BDI concepts, as displayed in Figure 4.4:

Actor A *Tropos* actor (we always refer to system sub-actors at architectural design) is represented as an *Agent Definition* defined in the Jadex Agent Definition language and containing the definition of its goals, plans and beliefs.

Goal Goals are mapped directly to Jadex goals. It is important to observe that a goal fails if no one of the plans that it triggers succeeds.

Plan Plans are mapped to Jadex plans. The mapping considers only those *Tropos* plans that have a direct means-end relationship to leaf goals. These plans can be seen as the attachment point for capabilities, implemented e.g. following the approach in [Penserini et al., 2006b]. In the present work these plans will be called *real-plans*, to distinguish them from the several other Jadex plans which are necessary in the following mappings to activate new goals at run-time (cf. Figure 4.2).

Resource Resources naturally map to an entry or a set of entries in the belief base. Since in Jadex the belief base corresponds to an object-oriented database, the entry can be related to an arbitrary Java object.

Softgoal Softgoals are not directly representable by BDI concepts. In *Tropos4AS* they are mainly used to define opportunities for the selection of alternative goals or plans to pursue along the goal model. A softgoal is therefore mapped to a belief base entry containing its name and a value that expresses the actual importance, which may change during runtime.

Contribution Contribution relationships to softgoals are therefore also stored only in the belief base and have no explicit representation by BDI concepts.

AND-decomposition To fulfil an AND-decomposed goal, all its subgoals have to be dispatched and finally achieved with success. As illustrated in left-hand side of Figure 4.5, the following solution was adopted: an AND-decomposed goal is set as trigger for exactly one plan, called AND-dispatch-plan (green hexagon). In the plan body, all subgoals will be dispatched in (some, perhaps random) order. If one subgoal fails, the process is stopped and the goal fails. No techniques to attempt compensation of already executed actions, have been considered for now.

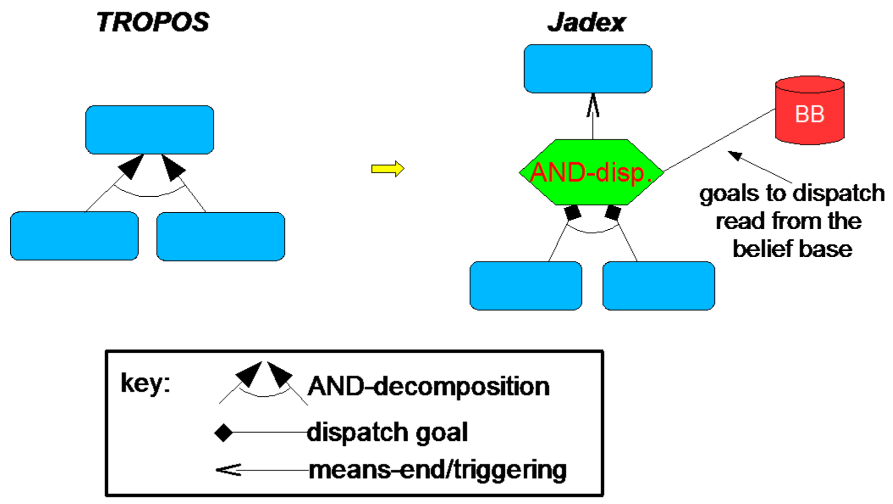


Figure 4.5: Mapping of the *Tropos* goal AND-decomposition into an equivalent Jadex BDI structure.

Means-end The *Tropos* means-end relationship can be mapped one-to-one to the Jadex plan triggering mechanism (Figure 4.6). Every time the associated goal is activated, plan execution is triggered. Similar to *Tropos* and *i**, Jadex supposes that every applicable plan for a goal is able to satisfy that goal completely. Therefore, for the case that more than one plan is available to fulfil the goal, the agent needs to be able to reason about which is the most convenient at that time. That is, if more than one plan is applicable for an active goal, a *meta-reasoning* process is adopted: a so-called *metagoal* is dispatched, which triggers an associated plan, the *metaplan*, that implements a strategy (e.g. some AI technique) to select between applicable plans. According to the semantics of the goal model, this algorithm should select alternatives, maximizing contribution to softgoals. A goal fails if none of the applicable plans can be executed with success.

OR-decomposition To achieve an OR-decomposed goal with success, one of the modelled alternative sub-goals has to be activated and finally to succeed. If the first subgoal fails (note that it can fail also due to a time-out), another one will be activated. The decomposed goal will fail only if no one of its subgoals succeed.

As previously seen, in the adopted BDI language, goals cannot activate other goals, but only be the triggering event for a plan. So, to map this kind of decomposition, Jadex plans have to be placed between goals and the OR-decomposed sub-goals, as illustrated in Figure 4.7. One of these *dispatch-goal plans* (hexagon) is triggered on the activation

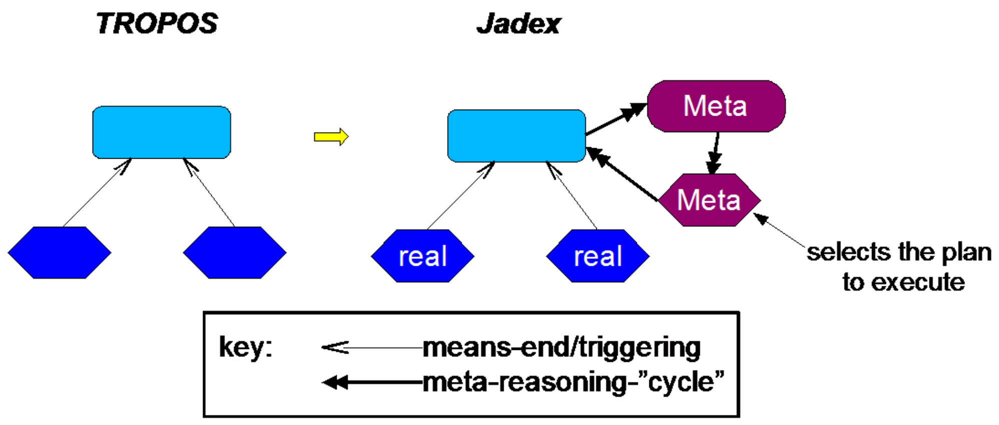


Figure 4.6: Mapping of the *Tropos* means-end relationship into an equivalent Jadex BDI structure.

of the parent goal and dispatches the relative subgoal. Since an OR-decomposition deals with at least two goals (and related plans) as alternatives to fulfil the triggered goal, the agent needs to be able to reason about which is the most convenient at that time. Since this mapping uses the same structures as the means-end relationship, the same meta-level reasoning mechanisms will be adopted.

Dependency As depicted in the upper part of Figure 4.8, if a dependency for a goal, plan or resource exists to fulfil a goal, the dependent actor relies on the dependee actor to deliver on this and to achieve or provide it. As our design framework mainly focusses on an actor representing the software system to be developed and its decomposition, we presume to obtain cooperative, usually closed multi-agent systems. Therefore, questions on trust and commitments, as dealt with e.g. in [Giorgini et al., 2005a, Chopra et al., 2010], are not considered in our work.

A flexible implementation of *Tropos* dependency relationships between two agents has to involve some form of interaction, which does not restrict the single agent's autonomy. For the mapping we involve the agent's messaging functionalities. In particular, we adopt a FIPA-standard request-interaction protocol. Figure 4.8 shows the *dependent* agent making a request to the *dependee* agent, indicating the *dependum*, i.e. the goal to be fulfilled. On the Jadex side, to the goal that is the cause of the dependency, a plan has to be associated, which initiates the request protocol.

Its counterpart, on the side of the dependee, is realized by a plan, (*request-plan*) triggered by that message. It informs the dependee (which could be, however, not only a software agent in the system, but also any external or human actor) for acceptance

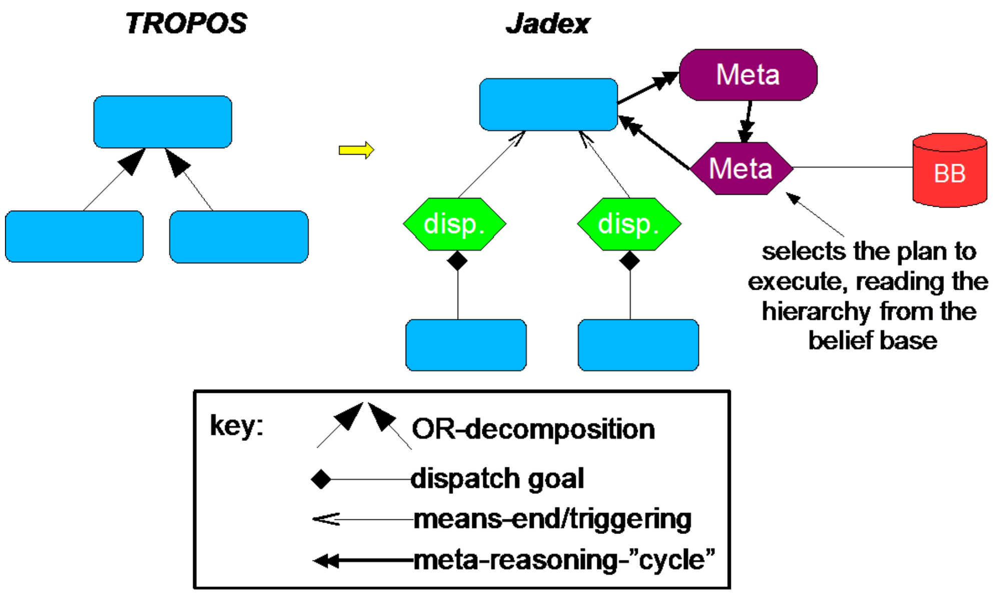


Figure 4.7: Mapping of *Tropos* goal OR-decomposition into the corresponding BDI structure.

fig:meansEnd

or rejection of the request, eventually dispatches the requested goal and finally communicates success or failure regarding goal fulfilment. For retrieving the agent that is able to fulfil a request, a yellow pages service, such as the Jadex Directory Facilitator can be utilized. See the FIPA specifications [O'Brien and Nicol, 1998] for details.

Enabling navigation of a goal model at run-time is important for any reasoning activity on top of it. The representation in the Jadex Agent Definition language describes the whole goal model and enables thus, theoretically, also its navigation. However, the Jadex runtime framework allows to navigate only the triggering links from plans to goals and does not provide any access from goals to plans. On the contrary, one of the key aspects of our framework is to enable the navigation of goal hierarchies, in order to improve the agent's decision making process. To make possible the navigation of the goal model, the concepts and structures of the goal model are additionally stored in the belief base.

4.2.4 *Tropos4AS* concepts mapping

In the following, a mapping for the new concepts introduced with the *Tropos4AS* framework is provided.

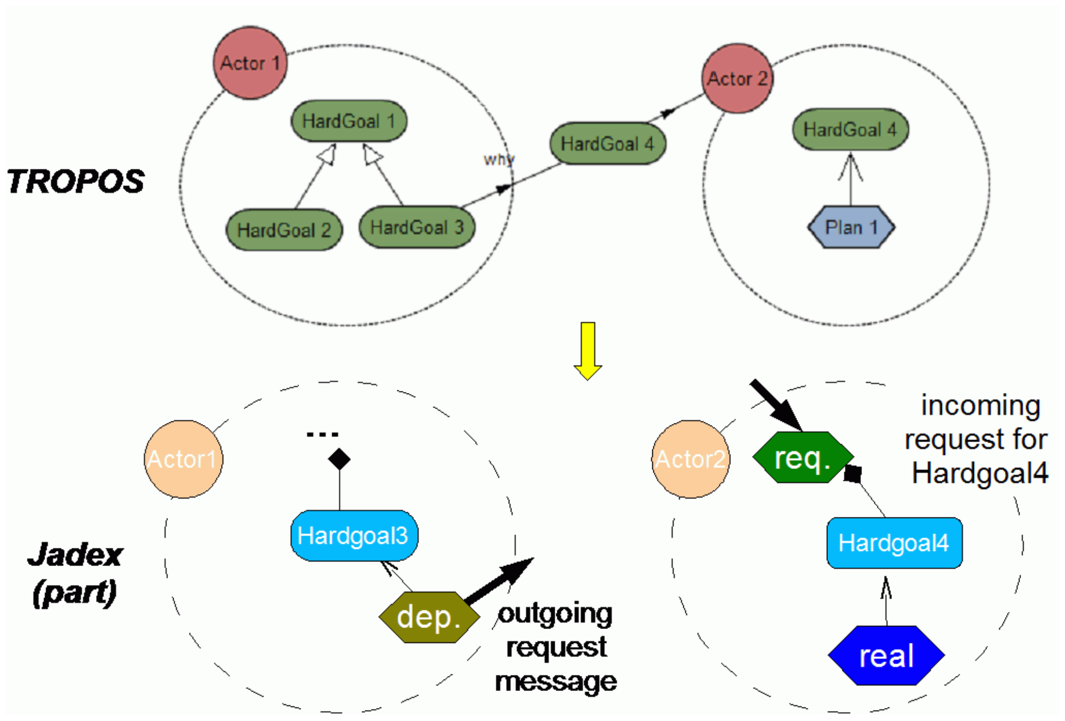


Figure 4.8: Mapping of the *Tropos* dependency relationship into the corresponding BDI structure.

The environment Artifacts of the system (and for the purposes of building a prototype, also artifacts of the environment, outside the system boundary) define the agent's perception of these objects and are thus mapped to facts in the agent's belief base. These artifacts, represented by simple UML class diagrams, are also mapped to Java classes providing an interface to the requested functionalities.

Extended goal models Goal types, defined in Section 3.2.2 are annotated in the corresponding goal in the BDI model. The different types of conditions are also mapped to the goal definition, using boolean formulas to link the goal achievement process to facts in the belief base, which represent the artifacts in the environment. It has to be noted that the definition of conditions, specifically of achievement conditions, for non-leaf goals, can change the goal achievement semantics. Details can be found in Chapter 5. Goal sequence is annotated in the belief base, whereas the *inhibition link*, a concept available in the Jadex language, can be directly mapped to the goal definition in the BDI model.

Failure models In *Tropos4AS*, a *failure* is a requirements analysis concept used mainly for eliciting missing agent capabilities and the conditions for their application. Thus, failures will not be directly represented in the BDI model.

Recovery activities are represented by pieces of ordinary goal models and are thus mapped together with the main goal model of the agent. These activities will appear in the agent, depending on the decision made in modelling step F4 in Section 3.3.2. In the case recovery activities are part of the main goal model, they will be part of it also in the implementation.

Alternatively, the recovery activities are kept as "loose" parts of the goal model structure within the agent definition. Errors are mapped to an achievement goal corresponding to achievement of the respective non-erroneous state, and defining the original error condition as its creation condition. For each plan or goal representing the entry point for a recovery activity, the goal representing the error is set as trigger, analogous to means-end relationships and OR-decompositions. In this way, the same meta-reasoning mechanism can be applied, if more than one recovery activity is available for an error in the current environmental context. To achieve this in an automated way, error conditions have to be formalized, regarding the interested environmental artifacts and the target language (Jadex/Java in our case). Such a mapping would also support the addition of new recovery activities at run-time, e.g. obtained by learning.

4.3 Tool Support

The visual modelling tool TAOM4E [4] was developed by the Software Engineering group at Fondazione Bruno Kessler (FBK), formerly ITC Irst, Trento [Perini and Susi, 2004, Morandini et al., 2008a]. It supports the *Tropos* modelling activities, originally from requirements analysis to architectural design. It is a plug-in for the Eclipse platform[5] and builds on the Eclipse EMF and GEF frameworks. TAOM4E implements the *Tropos* metamodel, as defined in [Susi et al., 2005], and allows various views on this model.

Actor diagrams can be graphically created and extended for the *Tropos* early and late requirements analysis phases. Actors can be detailed in a goal diagram, which is graphically shown in a "balloon" associated to the actor. In this balloon, delegated goals are visualized and can be decomposed. The actor representing the software system can

[4]Tool for Agent Oriented visual Modelling for the Eclipse platform, downloadable, including the presented extensions, at `http://selab.fbk.eu/taom`.

[5]Eclipse is an plug-in based, multi-platform, open-source development environment. See `http://www.eclipse.org` for details and download.

be detailed to a multi-agent system, in an “architectural design diagram”.

Figure 4.9 shows the principal parts of the tool front-end. The largest window, on the right hand-side, supports models editing according to the *Tropos* graphical notation that is provided by the *Palette* window (in the centre). Visualized diagrams are often partial views on the whole model. In the left-hand side window in Figure 4.9, titled *Navigator*, the folders with the output of the ***t2x*** code generation tool are shown, for each actor in the system, here Web Server, Search Actor, and Exam Parser.

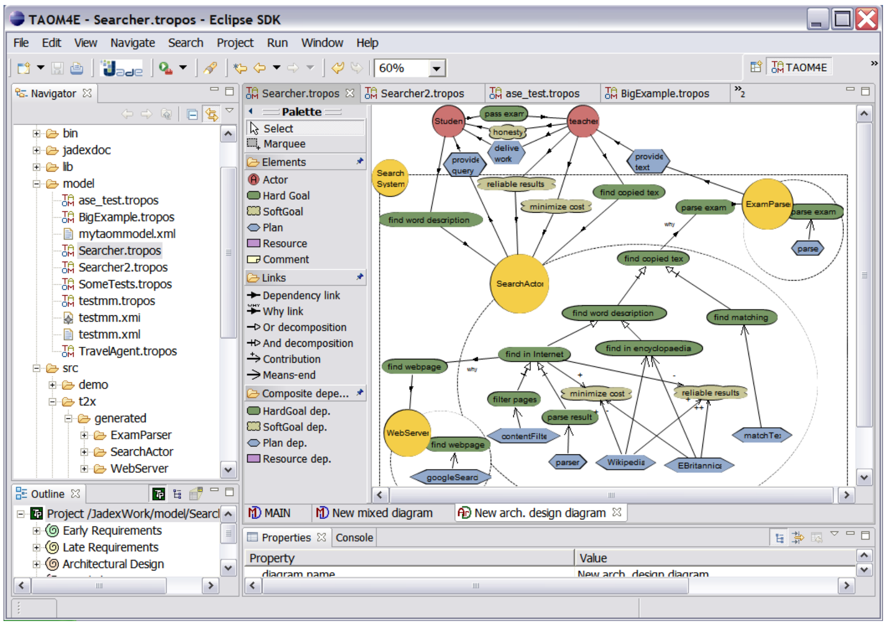


Figure 4.9: Interface of the TAOM4E Eclipse plug-in.

4.3.1 Tool support: *Tropos4AS* modelling

For the aim of supporting *Tropos4AS* with a modelling editor, the contribution of this thesis consists in extending TAOM4E along the following directions: the definition of the environment which surrounds the system; the introduction of goal types and goal relationships (i.e. inhibition and sequence); and the definition of conditions to correlate goal fulfilment with the environment.

The environment metamodel, which stores all relevant data for environment entities and conditions, and its graphical editor were built with the EMF/GMF framework. The extended TAOM4E tool provides a modelling editor to create and manage environment models, and an interfaces to manage goal conditions and relationships in TAOM4E diagrams. Furthermore it provides the generation of UML class diagrams from the environment model.

The created UML class diagrams can be edited with the Eclipse UML tools. UML-compatible (commercial) tools can also be used to import and edit them and to generate Java code and interfaces. We gained positive experience with the EclipseUML and IBM Rational System Architect tools.

4.3.2 Tool support: code generation

The code generation tool ***t2x*** takes in input *Tropos4AS* models created with the extended TAOM4E tool and produces a Jadex agent definition which adheres to the mapping guidelines presented in Section 4.2. A first version of this tool, limited to *Tropos*, was presented in [Morandini, 2006]. The generated agent prototypes are characterised by a BDI architecture and can be executed on the Jadex agent platform[6]. The generated code includes an explicit representation of the goal model and implements a basic message handling and goal achievement behaviour corresponding to the source goal model, and a failure-avoiding behaviour that exploits the modelled variability. The contribution by this tool is threefold:

First, it creates a representation of the goal model at run-time, which is navigable, monitorable and modifiable and gives thus a solid basis for an implementation of a self-adaptive system, which would possibly include complex learning and reasoning techniques, basing on this goal model.

Second, it enables the support for goal AND/OR hierarchies in a BDI architecture.

Third, it automatically generates agent code from goal models, enabling a fast and simple development of Jadex agent prototypes with the high-level behaviour defined in the requirements model. By strongly reducing the coding effort, this also facilitates the development of agent systems for students and non-expert programmers.

The concepts and structures identified in Section 4.2.2 have an almost direct correspondence in the Jadex Agent Definition Language, coded into an ADF (Agent Definition File) with XML format. Figure 7.9 and 7.10 in Section 7.2 (page 144) show a

[6]See Section 2.4.2 for a brief introduction on the features of the Jadex framework which are important for this work.

generated directory structure with Java file skeletons, a piece of generated ADF code, and illustrate some relationships between a goal model and this code.

An agent's goal model is coded in Jadex by mapping the goal decompositions to Jadex goals, as defined in Section 4.2, along with plans building the connection between goals at different levels, and a middleware (based on Jadex Plan classes coded in Java and accessing the model via the provided API) managing the decomposition logic.

For each *Tropos* plan, a Java file skeleton is generated and connected to goals in means-end relationship, by the Jadex *triggering* mechanism. To enable navigation, the goal decomposition graph is also stored in the agent's belief base (Figure 4.10).

```
<plan name="realPlan_Plan_2">
  ...
  <trigger>
    <goal ref="MyGoal_1"/>
  </trigger>
</plan>

<beliefset name="meansend" class="TLink">
  <fact>new TLink("MyGoal_1", "Plan_2")</fact>
</beliefset>

<expression name="query_ME_link">
  select $link.get(1) from TLink $link in $beliefbase.meansend
         where $link.get(0).equals($component)
  <parameter name="$component" class="String"/>
</expression>
```

Figure 4.10: Excerpt of an ADF showing the definition of a plan triggered by a goal, the representation in the belief base and a pre-defined model navigation query.

This architectural choice makes the agent aware about its abilities, namely, at run-time the agent can monitor and control its behaviour by navigating the modelled goal graph, to select goals and plans according to the modelled requirements. Softgoals and contributions to them are solely represented in the agent's beliefs. Reasoning algorithms can navigate them efficiently by pre-implemented belief base queries. Furthermore, ***t2x*** generates code for resolving dependency links between agents by the use of FIPA messaging protocols.

Artifacts in the system (for the purposes of building a prototype, also artifacts in the environment outside the system boundary) define the agent's perception of the represented objects and are thus mapped to facts in the agent's belief base, which can be accessed from the ADF as well as from Java classes through the API delivered with Jadex.

Run-time behaviour of generated prototypes

The generated agent prototypes are immediately executable, showing a basic goal-directed behaviour. At run-time, agents select between the alternatives modelled (OR-decompositions and means-end relations), based on the evaluation of *Tropos* contributions to softgoals, giving preference to goals and plans by maximizing contribution to softgoals. An immediate comparison between softgoals is obtained by defining an *importance* value for each softgoal, changeable also at run-time.

The default meta-reasoning plan selection method provided by Jadex is replaced in the prototype implementation by a simple depth-first tree traversal algorithm, calculating the utility of each alternative, by analysing contributions and softgoal importance. Starting at the interested goal and going until the plan level, it searches the path with the maximal sum of the products of contributions with softgoals importance. The qualitative *Tropos* contributions (+,++,...) are substituted by values in $[-1, 1]$, such that positive contributions will give higher, and negative contributions lower utility to an alternative.

The prototype implementation includes basic self-adaptation mechanisms obtained by evaluating environmental conditions and by exhibiting simple failure handling skills, taking full advantage of Jadex mechanisms: Alternatives included in the goal model are automatically explored if the execution of a plan or the fulfilment of a sub-goal fails. If all applicable plans fail (and therefore no more plans are applicable), a Jadex goal fails. These mechanisms is entirely handled by Jadex.

The generation tool was developed on purpose to obtain code which is easy to customise and to extend, not only in the declarative part of the agent definition, but also providing interfaces for the customization of the goal decomposition and reasoning mechanisms. However, this work limits to modelling at an agent's knowledge level, and the code does not contain the concrete activities the system has to be able to perform, which have to be implemented manually or by following proposals such as *capability modelling* [Penserini et al., 2006b].

Mapping details, illustrated on an example

Referring to the CleanerSystem example, presented in Section 7.1 (Figure 7.4), we show some details of the mapping carried out by the ***t2x*** tool. Figure 4.11 shows an example code generated for a goal of type *achieve*, which triggers the execution of associated plans, until reaching the defined target (i.e. achievement) condition. In this example, the goal EmptyFullDustbox has run-time precedence over the goal CleanField and is achieved when the agent believes that the dustbin is empty (i.e. the dustbin artifact reports the respective sensor information).

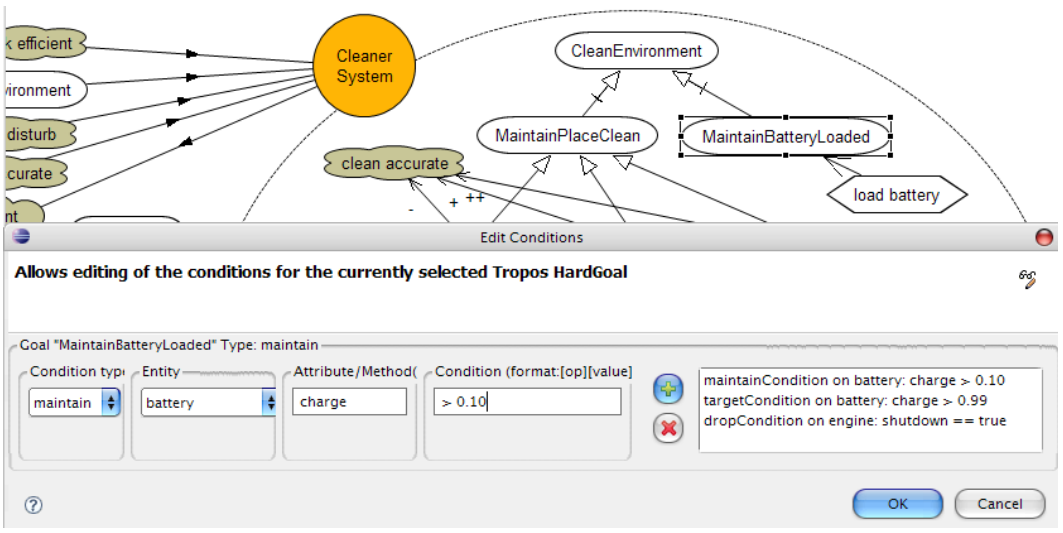
```
<achievegoal name="EmptyFullDustbox">
  <targetcondition>
    $beliefbase.dustbin.empty()
  </targetcondition>
  ...
  <deliberation>
    <inhibits ref=''CleanField''/>
  </deliberation>
</achievegoal>
```

Figure 4.11: Example of a goal definition in the Jadex ADF.

A goal of type *maintain*, MaintainBatteryLoaded, can be mapped straight-forward to a Jadex *MaintainGoal*. However, the correspondence between a *Tropos4AS* goal and a goal in the Jadex language is not always straight-forward. For example, the behaviour of a *perform-goal*, e.g. FindCoarseDust, does not correspond to the homonymous Jadex goal type, because at execution time associated plans would be executed only once. The appropriate goal fulfilment behaviour we expect for a perform-goal, denoted as *perform goal with retry-flag* in [Dastani et al., 2006], corresponds to a *RecurrentPerformGoal* in Jadex. The different types of conditions, defied in Section 3.2.2, are mapped to the ADF, using boolean formulas to link the goal achievement process to facts in the belief base, which represent the artifacts in the environment and are implemented in JAVA classes containing the methods that represent the functionalities of the artifacts. Goal sequence in an AND-decomposition is annotated in the ADF and implemented in the Java plans defining the AND-goal decomposition logic. The *inhibition link*, a concept available in the Jadex language, can be directly mapped to an XML tag of the form `<inhibits ref="goal_to_inhibit"/>` for the inhibiting goal. Further details on the mapping and code generation, limited to *Tropos*, can be found in [Morandini, 2006].

4.3.3 Tool architecture

TAOM4E, its *Tropos4AS* modelling extensions, and the ***t2x*** code generation tool are developed as three plug-ins for the Eclipse framework. The meta-models for the editors were defined with the Eclipse Modelling Framework (EMF). The graphical model editor for TAOM4E was developed with the aid of a preliminary version of the Eclipse Graphical Editor Framework (GEF) and heavy manual modifications on the generated code. An integration of new graphical modelling elements into the existing TAOM4E diagram editor would thus have required an unproportionately high programming effort and a partial reprogramming of the editor.



Allows editing of the conditions for the currently selected Tropos HardGoal

Goal "MaintainBatteryLoaded" Type: maintain

Condition typ	Entity	Attribute/Method(	Condition (format:[op][value]
maintain	battery	charge	> 0.10

maintainCondition on battery: charge > 0.10
targetCondition on battery: charge > 0.99
dropCondition on engine: shutdown == true

Figure 4.12: Taom4E: Interface for defining conditions on modelled entities.

We thus decided to implement an own plug-in with an editor for the environment model and support the modelling extensions, basing on the Eclipse Graphical Modelling Framework (GMF). The underlying model for the editor holds not only the environment diagram, which is graphically visualized, but also an own representation of the goal model able to hold additional information on to the extensions. To model additional details within TAOM4E, this plug-in links into the context menus of the TAOM4E diagram editor, providing e.g. an interface for defining goal conditions on modelled environmental entities (visulalized in Figure 4.12) and for additional relationships, direct access to the environment model of each agent, and the generation of UML class diagrams.

Figure 4.13 summarizes the layers of the tool architecture, including the ***t2x*** tool which takes the models created by both editors in input, generates a Jadex Agent Def-

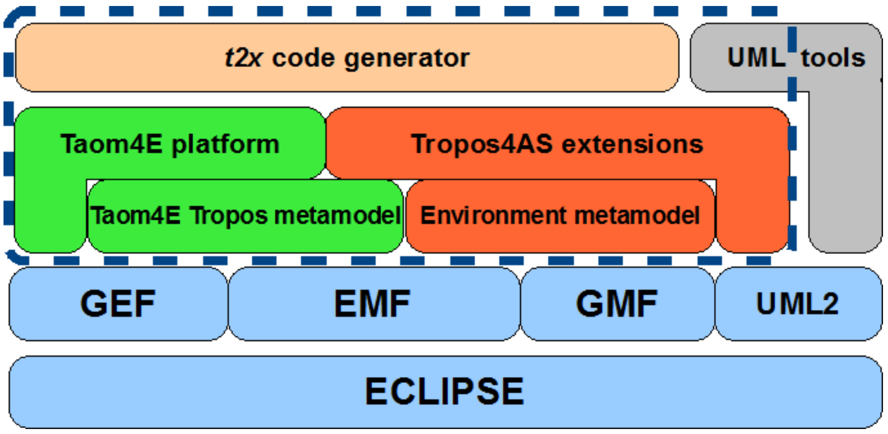


Figure 4.13: Taom4E and *t2x* tool architecture

inition File which represents the extended goal model and supporting Java structures and prepares the run-time environment. The three plug-ins are installable through the Eclipse Update Manager, from the Update Site `http://selab.fbk.eu/taom/eu.fbk.se.taom4e.updateSite/`. Currently, the TAOM4E tool is re-engineered by the use of GMF with model extendibility as primary requirement. This new version would enable the integration of extensions, environment and failure models into a single graphical model editor.

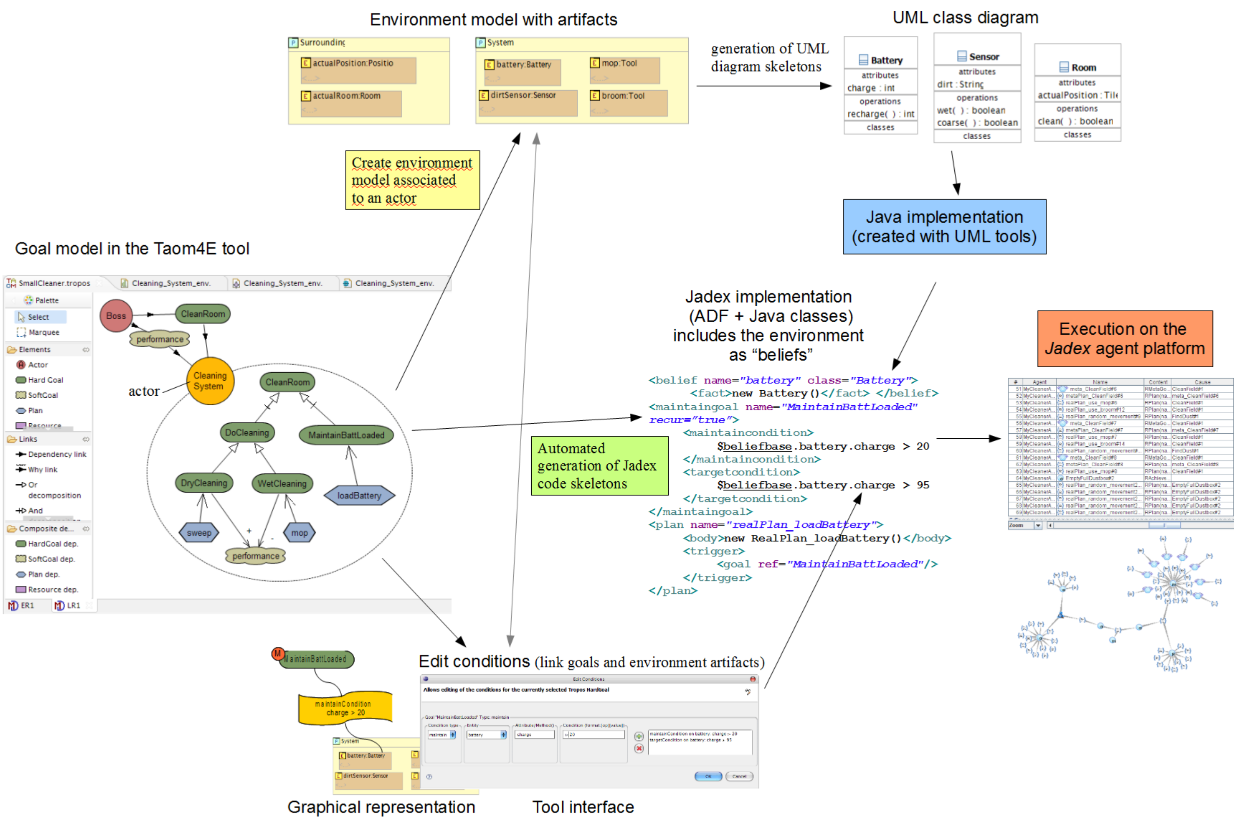
4.3.4 Illustration of the tool-supported process

We now assemble the various tools involved in *Tropos4AS* and illustrate the process described in Section 4.1 and Figure 4.1, through their application.

The *Tropos* goal model, drawn with the TAOM4E tool for the early and late requirements analysis phase, and then (if desired) decomposed to sub-actors in an Architectural Design Diagram, can be at any time extended, by creating an environment model for a system actor (in the actor's context menu). The environment model holds a simple representation of the artifacts of interest for the actor, grouped in packages.

Failure modelling is currently not supported by these tools. If a representation and a mapping to agent code for errors and related recovery activities is desired, errors can be represented by goals for achieving an error-free state, highlighted by a pale red colour, which are activated by the error condition. Recovery activities are described by goal model fragments, linked to this goal by a means-end relationship.

The *Tropos4AS* extensions, including conditions, can subsequently be defined in the interfaces provided for every goal and plan in the diagram (left part of Figure 4.14).


Environment model with artifacts
UML class diagram
generation of UML diagram skeletons
Battery
charge : int
recharge( ) : int
Sensor
dirt : String
wet( ) : boolean
coarse( ) : boolean
Room
clean( ) : boolean
Create environment model associated to an actor
Java implementation (created with UML tools)
Goal model in the Taom4E tool
Boss
CleanRoom
performance
actor
Cleaning System
DoCleaning
MaintainBattLoaded
DryCleaning
WetCleaning
loadBattery
sweep
mop
Jadex implementation (ADF + Java classes) includes the environment as “beliefs”
Automated generation of Jadex code skeletons
Execution on the *Jadex* agent platform
Edit conditions (link goals and environment artifacts)
maintainCondition charge > 20
Graphical representation
Tool interface

```
<belief name="battery" class="Battery">
     <fact>new Battery()</fact> </belief>
<maintaingoal name="MaintainBattLoaded"
recur="true">
     <maintaincondition>
          $beliefbase.battery.charge > 20
     </maintaincondition>
     <targetcondition>
          $beliefbase.battery.charge > 95
     </targetcondition>
</maintaingoal>
<plan name="realPlan_loadBattery">
     <body>new RealPlan_loadBattery()</body>
     <trigger>
          <goal ref="MaintainBattLoaded"/>
     </trigger>
</plan>
```

Figure 4.14: The Tropos4AS process, illustrated through the application of the modelling tools.

Having concluded the modelling activities, by the ***t2x*** tool, Jadex ADF files and supporting Java code for agent prototypes can be generated. The class diagrams representing the environment can be edited by available UML tools (e.g. EclipseUML, upper left part of Figure 4.14), which also give the possibility to generate Java interfaces, accessible by the implemented agent prototypes.

The ***t2x*** Eclipse plug-in also provides support for an in-place execution of instances of the generated agents, on the Jadex agent platform. The Jadex *Introspector* tool provides various visualizations of the agent's internals during execution. The molecule-like graph displayed in the right part of Figure 4.14 graphically shows the activation of goals and execution of plans for an agent, which can be directly related to its design-time goal model (see Figure 4.15 for an example).

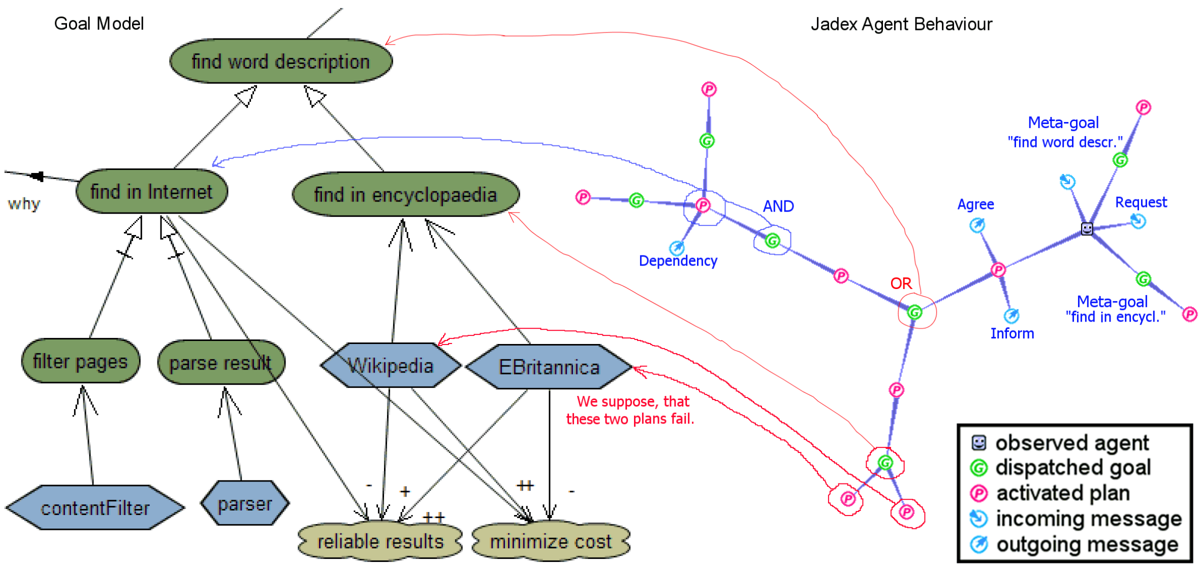


Figure 4.15: Goal model fragment for a simple example and relationships to the agent's execution (graphical run-time representation by the Jadex Introspector tool), for code generated by the ***t2x*** tool (with the only manual modification of defining the failure of two plans, for visualizing the transversal of the whole model).

4.3.5 Discussion

The mapping and implementation which we propose, is one out of many possible, and in defining it we had to make inevitably several decisions and simplifications. The implementation gives a general run-time environment for the interpretation of the knowledge represented in the models. Adaptivity is thus limited to basic behaviours. However, the obtained code gives a foundation which can be customised, either centrally at an agent level, or at the level of the single decision points (e.g. extending meta-level reasoning algorithms), with specific, more sophisticated or domain-dependent mechanisms for monitoring, for decision-making and for performing the adaptation.

Moreover, several problems emerge when dealing with agents at run-time, related to reliability, security, scalability, and performance. The developed prototype is mainly used as a feasibility study, so security and performance problems were not addressed. However, the mapping is linear and the messaging functionalities are provided by the agent platform, thus these quality requirements are highly dependent from the underlying platform. Focusing on reliability and failure handling, non-terminating subgoals or plans are an issue for the goal satisfaction process. The same problem arises when goals are delegated to other agents and thus one agent is dependent from the other.

Time-outs were defined in the implementation to prevent deadlocks in such cases. To address scalability, we suggest to give particular attention to the decomposition of the system to sub-agents.

A remaining issue is related to the maintainability of the code. Despite having separated the single functionalities from the knowledge level, the behaviour of the implemented system soon becomes complex, because of the various threads of control emerging from the contemporaneous satisfaction of the different goals (especially maintain-goals) at run-time, and their synchronisation with the monitoring functionalities.

4.4 Final Considerations and Related Work

In this chapter, the conceptual models and modelling steps introduced in the previous chapter were integrated to a complete, tool-supported process, which spans the development phases until the implementation, focussing on knowledge-level artefacts. This process includes a mapping from the design artefacts to a BDI agent architecture, to preserve the representation of the high-level requirements in form of a goal model at run-time, and to lower the conceptual gap between software requirements, design and implementation.

Thus, our approach differentiates from other agent-oriented methodologies such as Prometheus [Padgham and Winikoff, 2002], MaSE [DeLoach et al., 2001] and ADELFE [Bernon et al., 2005]. These methodologies use goal models for capturing the high-level requirements, but change the design focus in the later steps from goals to tasks, messages and data, loosing the concept of goal. [Khallouf and Winikoff, 2009] refines Prometheus, maintaining goals until the implementation, but looses the goal model structure. With the same consequence, in O-MaSE a goal model is implemented by assigning each leaf goal to a single agent role in a MAS [Oyenan and DeLoach, 2010].

For a concrete implementation we adopted the Jadex BDI agent platform, which provides a reasoning cycle and an explicit representation of goals at run-time. The *Tropos4AS* goal hierarchy is mapped to a BDI agent structure, by mapping hierarchical goal models to Jadex goals along with plans which contain the decomposition logic and build the connection between goals at different levels of the hierarchy. The obtained run-time behaviour reflects the intended semantics attributed to the goal model at design time, preserving variability and defining the decision criteria for alternatives selection. The approach does not include the detailed modelling and implementation of the single functionalities, which could however be implemented by the use of traditional engineering techniques.

In comparison, [Nakagawa et al., 2008] maps goals to *behaviours* of JADE agents, thus missing a goal-directed behaviour and an explicit representation of the goal model. A mapping from *Tropos*, extended with temporal execution annotations, to Prolog [Cares et al., 2005] also looses the goal model structure at run-time. [Salehie, 2009] represents and traverses the goal model at run-time. The aproach in practice rebuilds parts of a BDI architecture for goal activation and deliberation. On the contrary, *Tropos4AS* relies on an available BDI architecture with its goal deliberation mechanisms. [Krishna et al., 2006] defines a mapping from *i** models to 3APL agents. Softgoals and dependencies are mapped similar to our approach. However, the problem of selecting between alternatives is not addressed and it is not defined how softgoals would influence rule selection. A direct mapping, without any middleware layer, was achievable, since the mapping bases on the original *i** [Yu, 1995], in which goal AND/OR decomposition is not specified. Also, no code generation tool is provided.

We provide tool support for conceptual modelling, by extending the TAOM4E development environment for supporting *Tropos4AS* modelling concepts. The ***t2x*** (Tropos4AS to Jadex) code generation tool provides an automated mapping from *Tropos4AS* goal models to BDI agents executable on the Jadex platform, including a middleware for the navigation and decision making on the goal model at run-time. The generated prototypes have with a basic message handling and goal achievement behaviour, exhibiting a set of behaviours corresponding to the source goal model, including the ability to reason about alternatives to achieve goals. The generated code can be modified and customized as needed, i.e. adopting more sophisticated, domain-depended learning and reasoning techniques.

The mapping process can be straightforwardly adapted to other agent languages with a BDI architecture, whereas for non-goal-oriented languages, such as JADE and object-oriented programming languages, a middleware layer would be needed, implementing functionalities for goal monitoring and a run-time reasoning cycle. The implementation is also ready for run-time modification of a goal model, and thus of the agent's knowledge level behaviour, guided by the users, by supervisor agents, or by the agent itself, for instance exploiting machine learning techniques.

To further consolidate the *Tropos4AS* framework, the automated mapping towards a BDI architecture has to be tested on real-world applications, in which more complex behaviours have to be implemented. The ***t2x*** code generation tool has been used in university courses for an introduction to BDI agents programming, by various students for their course projects and in master theses.

Chapter 5

Operational Semantics for Goal Models

5.1 Introduction

Several agent-oriented software engineering methodologies address the emerging challenges posed by the increasing need of adaptive software. A common denominator of such methodologies is the paramount importance of the concept of *goal model* in order to capture and understand the requirements of a software system [van Lamsweerde, 2001, Borgida et al., 2009].

Various agent programming languages, such as JACK, Agentspeak[1] and Jadex incorporate the notion of *goal* as a language construct and give the possibility to define goal *types*. These types define different attitudes that agents can have towards their goals [Dastani et al., 2006]. A formalisation which unifies and formalizes goal types and connected conditions defined in various languages, is addressed by Riemsdijk et al. [van Riemsdijk et al., 2008]. However, this formalisation limits to goals which are directly operationalised by plans[2]. Moreover, goal models (i.e. goal hierarchies) are not natively available, to our knowledge, in any agent programming language.

The modelling language introduced with *Tropos4AS* allows to define goal models, enriched with information on the dynamic behaviour, defining goal types and associated conditions for goals at any level of decomposition. The interplay of these concepts

[1]AgentSpeak is the language interpreted by the *Jason* agent platform [Bordini and Hübner, 2005].

[2]We define goals which are directly operationalised by plans as *leaf-level goals* or *leaf goals*, while *non-leaf goals* are goals in a goal hierarchy, decomposed to sub-goals and operationalised only at a lower level.

calls for a formalisation of the behaviour intended when modelling them at design time. Also, a central aim of *Tropos4AS* is to preserve design-time knowledge on goals and variability until run-time, to take autonomous decisions for achieving high level objectives correctly. An implementation should thus respect this intended behaviour.

In this chapter we define an operational semantics for goals in a hierarchical goal model, building upon the semantics for "leaf-level" goals defined by Riemsdijk et al [van Riemsdijk et al., 2008]. First we define an abstract architecture, able to capture a large pool of possible goal achievement behaviours. Upon this, we formalise the semantics for achieve-goals, maintain-goals, and perform-goals, in goal models with AND- and OR-decompositions. Higher level (non-leaf) goals are defined through the achievement of their sub-goals and through the satisfaction of their achievement conditions. These semantics give a formal definition of the behaviour that can be expected from the execution of a goal model modelled at design time, by defining its run-time behaviour.

The chapter is structured as follows. Section 5.2 gives an intuitive idea of goal model semantics at run-time with the help of a simple cleaner agent example, Section 5.3 introduces the formalisation of our goal model semantics as an abstract architecture, on which, in Section 5.4 the behaviours for various goal types are instantiated. A discussion on the obtained results and an example for the application of the semantics conclude this chapter.

5.2 Goal Types and Goal Decomposition

In *Tropos4AS* we introduce expressive extensions to goals, to concretise the correlations between high-level requirements and system functionalities: goal types and their conditions. Goal types precisely detail the agent's life-cycle by defining the run-time behaviour for (i.e. the agent's attitude towards) achieving a goal. Conditions guide and guard state transitions in this life-cycle. Besides, *Tropos4AS* allows also to model goal AND/OR decomposition, and thus gives the possibility to model a wide spectrum of possible agent behaviours. This gives the need for a formal definition of the process of goal satisfaction, hereafter called the *goal life-cycle*, for goals embedded in a goal model.

The semantics for goals in goal models should cover the goal types typically available in a BDI-based agent language. Following the terminology introduced by Dastani et al. [Dastani et al., 2006], we consider the three main goal types *achieve*, *perform* and *maintain*. Moreover, the semantics have to consider goal AND/OR decomposition. Non-leaf goals are not directly pursued by executing plans, but by activating one or more of their subgoals. The satisfaction process is thus more complex, because two facts have to be assessed: the satisfaction of one (OR) or all (AND) subgoals and the satisfaction of conditions. Also, the definition of goal failure can depend on both the decomposition and specific conditions.

A prerequisite for the abstract semantics is the ability to cover the following **goal types**, characterised by a specific satisfaction behaviour and by various conditions:

An ***achieve-goal*** is characterized by an achievement condition that specifies when a certain state of affairs is reached. The satisfaction of the goal can be attempted several times till this condition holds. Moreover, a failure condition can terminate goal achievement, defining it as failed.

To satisfy a ***perform-goal***, the agent has to successfully execute some actions (plans), without demanding that the plans must reach the states denoted by the goal.

Last, for satisfying a ***maintain-goal***, the agent has to try to maintain a certain state of affairs. In literature, different types of semantics have been attributed to maintain-goals. E.g., an agent can act reactively or proactively to maintain a state [Duff et al., 2006]. In the first case (*reactive* maintain-goals), it starts taking action when a particular state is no longer maintained, while in the second case (*proactive* maintain-goals) it tries to act to *prevent* the failure of the maintenance condition. The implementation of proactive maintenance goals, although suitable for formal verification [Fuxman et al., 2001], would, at run-time, require predictive reasoning mechanisms, which are not easily representable through an operational formalisation, and in procedural, event-guided agent languages in general [van Riemsdijk et al., 2008].

Thus, in this work we focus on ***reactive maintain-goals***, which are available on most agent platforms. Such goals are *activated* each time their maintenance condition is not satisfied and *suspended* if the condition holds. Proactive maintain goals would theoretically also be modellable in our framework but ask for a predictive evaluation of maintain-conditions. This would demand the use of heuristics and reasoning techniques which is out of the focus of this work and would moreover not be successful in every domain.

5.2.1 An example: the Cleaner Agent

To illustrate how a goal model captures the intended run-time behaviour, we refer to a very simple cleaner robot scenario, which can be found in several variations in artificial intelligence and multi-agent systems fields.

The *Cleaner Agent*, modelled with *Tropos4AS* (Figure 5.1), represents the control software for an autonomous robot that could ideally be employed in an office building.

The *achieve-goal* `RoomClean` has an achievement condition "room clean at the end of the day" and is OR-decomposed into the two alternatives `DryCleaning` and `WetCleaning` (both are *perform-goals*). These "leaf-level" goals are operationalised by plans that give different contribution to the quality requirement *efficiency*, modelled as a softgoal.

Supposing both alternative subgoals are applicable in the current context, the agent will pursue the subgoal that maximizes contribution to its softgoal `efficiency`. The semantics of the goal model now allow designers to characterize various run-time behaviours of an agent, like the ones shown in the following scenarios.

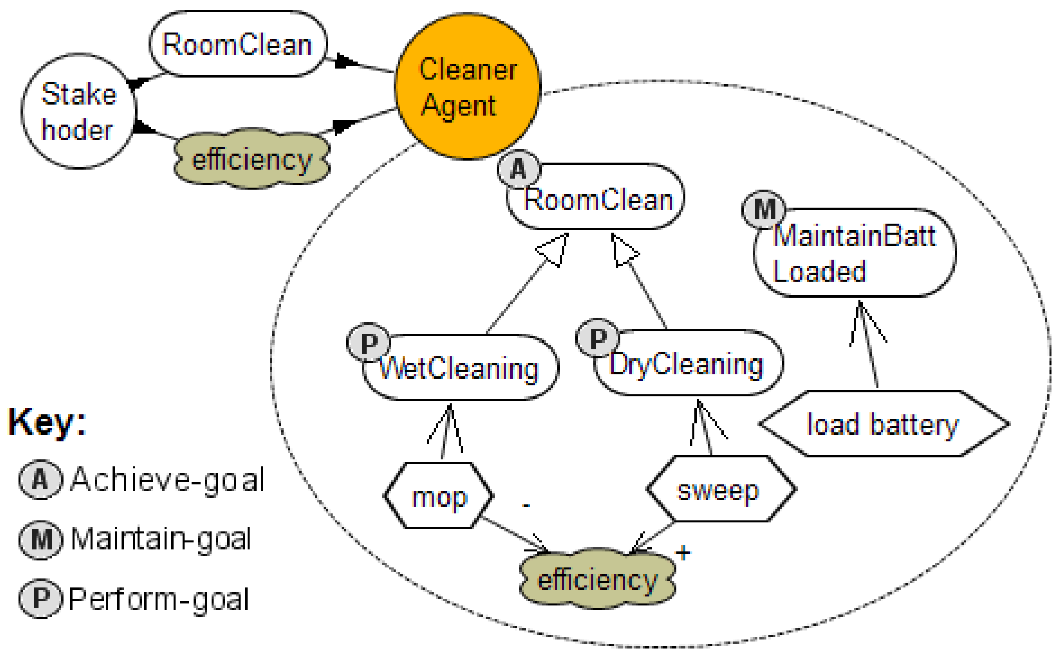


Figure 5.1: Fragment of a goal model for the Cleaner Agent example.

Scenario 1. The agent achieved `DryCleaning` by using a broom, but due to some stubborn dirt, the achievement condition of the main goal `RoomClean` is not yet satisfied. Thus, the agent should retry the other available alternative, the goal `WetCleaning`, hoping that after its achievement the main goal will be achieved.

Scenario 2. Suppose that the agent is cleaning the room with a mop, performing the goal `WetCleaning`, and runs out of water. If all the dirty parts of the floor were already cleaned (and the agent can sense this), the achievement condition of `RoomClean` is satisfied and thus, after all, the top goal succeeds.

Interesting considerations arise by modelling these scenarios, in which the agent's behaviour adheres not only to the semantics of goal AND/OR decomposition but is also driven by the nature of goal types along with their satisfaction conditions. For example, in Scenario 1, one of the two alternative subgoals for the main goal `RoomClean` was correctly performed. However, the state denoted by the achievement condition is not yet reached. On the contrary, in Scenario 2, the subgoal was not correctly performed, but nevertheless the top-level goal should succeed. The different parameters allow for various interpretations regarding the agent's behaviour.

5.3 Goal Model: Abstract Architecture

In the following, we provide formal operational semantics to deal with non-leaf goals in a goal model, and customize these semantics for each goal type, illustrating how they adhere both to the semantics of run-time goals (as in agent languages, by the use of conditions) and to the interpretation given to hierarchical goal decomposition in goal models of agent-oriented software engineering methodologies like *Tropos*, *Tropos4AS* and KAOS.

Taking ideas from the formalisation used in [van Riemsdijk et al., 2008], we first define an abstract goal architecture able to capture a large pool of possible goal achievement behaviours (Section 5.3), in order to instance upon this the desired run-time behaviour for various types of goals (Section 5.4).

The architecture defines the different states of a goal in the run-time goal satisfaction process, for AND- and OR-decomposed goals, and the operational semantics of goal satisfaction, in terms of transition rules, where some of the transitions are controlled by specific conditions and thus allow to customize the goal satisfaction process for various goal types.

5.3.1 Basic Concepts of the Formalisation

In the abstract architecture proposed by [van Riemsdijk et al., 2008], once adopted, (leaf) goals can have two different states: suspended and active. In the active state planning and execution of plans take place. The satisfaction process for non-leaf goals is more complex, essentially because two facts have to be assessed: the satisfaction of subgoals of AND/OR decompositions and the satisfaction of the conditions defined for a particular goal type. The flexible interplay between these two aspects calls for additional goal states to explicitly represent failure and success in the goal achievement process.

We define an abstract architecture for non-leaf goals, which includes the following goal states $S = \{$*suspended (S), active-deliberate (AD), active-undefined (AU), active-success (AS), active-failure (AF)*$\}$.

In the following we define transition rules between these states, labelled as *adopt, activate, suspend, deliberate, subgoal-achieve, fail, succeed, retry, reactivate, drop-failure,* and *drop-success* in Figure 5.2.

Some of these transitions are governed by *transition actions* (one of ACTIVATE, SUSPEND, FAIL, SUCCEED, RETRY, REACTIVATE, DROPFAILURE, DROPSUCCESS) which are correlated to specific transition conditions. In Section 5.4, these conditions will be instantiated to obtain the desired behaviour for the different goal types, "enabling" the proper actions.

The transition rules exactly define the five states. However, to clarify the idea behind these states, here we give a brief, intuitive description of them.

The *suspended* state (S) is, analogous to [van Riemsdijk et al., 2008], the state in which a goal is adopted, but the agent has to wait for the activation of it (e.g. due to some unsatisfied condition). Once activated, the goal transits to one of the four states which denote a state in which the agent actively acts for satisfying the goal. State *active-deliberate* (AD) is an intermediate state, which is reached directly after activation of a goal. Immediately, the agent tries to deliberate subgoals, i.e. to reveal the list of feasible subgoals of the current goal, and passes to *active-undefined* (AU). In this state the agent tries to adopt the revealed subgoals and eventually analyses their success or failure (differently for AND- and OR-decompositions).

Depending on the decomposition type, the goal will transit to the "provisional" failure or success states (AF) or (AS), depending on subgoal achievement. Transitions to (AF) and (AS) can also be guided by the satisfaction of the transition actions FAIL or SUCCEED (which will usually be bound to goal failure and success conditions). The same transition actions are re-evaluated in (AF) and (AS), to finally drop a goal with success or failure. However, in these states, transition actions (later instantiated to proper conditions) can also SUSPEND a goal (this is usually done for maintain-goals), restart its achievement process from subgoal deliberation REACTIVATE (e.g. to restart goal achievement if all subgoals succeed but the goal is still not satisfied), or RETRY with another subgoal, if available (e.g. if one alternative subgoal succeeds, but the goal is not yet satisfied and so another available alternative should be tried).

We aimed at minimizing the set of possible transitions. By a sophisticated combination of transitions all necessary state changes can be covered.

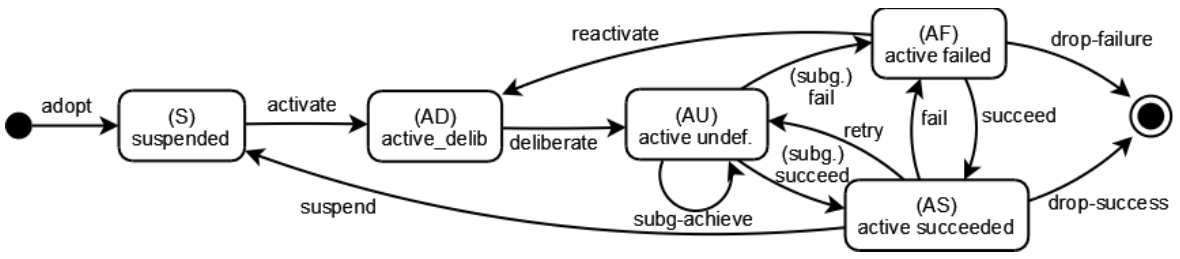


Figure 5.2: Possible states and transitions in the abstract architecture for non-leaf goals in goal models.

Definition 1 The state of an agent is characterised by a tuple $\langle B, G\rangle$, where: B is the agent's actual set of beliefs (the *belief base*), which contains a set of beliefs and known facts about the surrounding world, perceptions, messages and its internal state; G is the set of goals $\{g_1 \dots g_n\}$, the agent actually has to pursue, i.e. the *adopted* goals. $B \models c$ denotes that condition c is satisfied with respect to the actual set of beliefs B.

Definition 2 A generic non-leaf goal at run-time is defined as $g(C, E, s, \Gamma)$, where $s \in S$ is the actual goal state and Γ is a list of goals that results from a deliberation activity $deliberate(g, B)$, returning applicable subgoals for g: $\Gamma = \{\gamma_1 \dots \gamma_n\}$. C and E are tuples of the form $\langle condition, action\rangle$, where *action* is one of the transition actions previously defined and *condition* is evaluated in B.

Definition 3 A condition c in C is evaluated in B if $\Gamma \neq \emptyset$ (i.e. the set of adopted subgoals is not empty). A condition c in E is evaluated if $\Gamma = \emptyset$.

5.3.2 Transition rules

The operational semantics for our abstract architecture are defined by a set of inference rules that define possible state transitions. Each rule is specified as

$$\frac{L}{R} \qquad \textit{[rule-name]}$$

where R represents a possible state transition of the system under the set of conditions L.

In the following, we define the transition rules for goals in G, both for goal AND- and OR-decompositions in a goal model. We thus assume that the goals are already adopted, and thus initially in state S (suspended).

New goals can be added to G (*goal adoption*) upon request from outside, triggered by a creation (adoption) condition or as subgoals during goal achievement. Goal adoption which does not origin from a goal hierarchy, is not further detailed here. In brief, to guarantee that the control of subgoals is left to the parent goals in a goal tree, external (user) requests for the adoption of goals (in an agent architecture typically messages) and creation conditions should be allowed only for root goals.

Goal activation

Goal activation is guided by a condition c. The following two transition rules, *[activateC]* and *[activateE]*, define the state transition from the state S (suspended) to AD (active, deliberation), depending on the condition associated to the action ACTIVATE. The naming includes the transition labels in Figure 5.2; "E" and "C" denote that the transition is applied for an empty or non-empty set Γ, respectively. Unless otherwise defined, all rules are used for both AND- and OR-decomposition.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Activate}\rangle \in C \qquad B \models c}{\langle B, g(C, E, S, \Gamma)\rangle \rightarrow \langle B, g(C, E, AD, \Gamma)\rangle} \qquad \textit{[activateC]}$$

$$\frac{\langle c, \textsc{Activate}\rangle \in E \qquad B \models c}{\langle B, g(C, E, S, \emptyset)\rangle \rightarrow \langle B, g(C, E, AD, \emptyset)\rangle} \qquad \textit{[activateE]}$$

Subgoal achievement

The first step in non-leaf goal achievement consists in revealing its subgoals. For this, the function *deliberate* returns a list Γ of subgoals to satisfy, while the goal state changes from AD to AU. In its simplest form, the deliberation function returns the whole set of subgoals, but also complex algorithms for subgoal discovery could be implemented. No deliberation takes place in the case that there are still subgoals available, i.e. $\Gamma \neq \emptyset$[3]. Note that *[deliberateE]* has no pre-conditionis and is thus an axiom.

$$\frac{}{\langle B, g(C, E, AD, \emptyset)\rangle \rightarrow \langle B, g(C, E, AU, \mathit{deliberate}(g, B))\rangle} \qquad \textit{[deliberateE]}$$

$$\frac{\Gamma \neq \emptyset}{\langle B, g(C, E, AD, \Gamma)\rangle \rightarrow \langle B, g(C, E, AU, \Gamma)\rangle} \qquad \textit{[deliberateC]}$$

[3]This particular transition would be required for temporal goal suspension, which is not further detailed here.

At this point, subgoal adoption (and thus, eventually, their achievement) can take place. AND- and OR-decomposed goals have different achievement semantics. Intuitively, the goal remains in the undefined state AU as long as the result of subgoal achievement is uncertain. Thus, an AND-decomposed goal remains in AU until one subgoal fails (rule *[AND:subg-achieve]*), in which case it will change to the "provisional" failure state AF *[AND:subg-fail]*. When all subgoals are pursued ($\Gamma = \emptyset$) and the goal is still in state AD, applying *[AND:goal-succeed]* it will transit to the "provisional" success state AS[4].

Referring to OR-decomposition, a goal transits to AS at the first success of a subgoal, and to AF only if all subgoals fail. Each instance of a subgoal γ updates the belief base with $success(\gamma)$ or $failure(\gamma)$, depending if it was achieved or not. Accordingly, our formalisation provides this information to the belief base when a goal is dropped. To ensure that transitions triggered by true conditions have precedence over adopting a new subgoal, the next four transition rules also need the precondition $\neg\exists\langle c, a\rangle \in C\ .\ (B \models c) \wedge a \in \{\textsc{Fail},\ \textsc{Succeed}\}$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G,\gamma_i)\rangle \to \langle B', G\rangle \qquad B' \models success(\gamma_i)}{\langle B, g(C,E,AU,\Gamma)\rangle \to \langle B', g(C,E,AU,\Gamma\setminus\{\gamma_i\})\rangle} \qquad \textit{[AND:subg-achieve]}$$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G,\gamma_i)\rangle \to \langle B', G\rangle \qquad B' \models failure(\gamma_i)}{\langle B, g(C,E,AU,\Gamma)\rangle \to \langle B', g(C,E,AF,\Gamma\setminus\{\gamma_i\})\rangle} \qquad \textit{[AND:subg-fail]}$$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G,\gamma_i)\rangle \to \langle B', G\rangle \qquad B' \models failure(\gamma_i)}{\langle B, g(C,E,AU,\Gamma)\rangle \to \langle B', g(C,E,AU,\Gamma\setminus\{\gamma_i\})\rangle} \qquad \textit{[OR:subg-achieve]}$$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G,\gamma_i)\rangle \to \langle B', G\rangle \qquad B' \models success(\gamma_i)}{\langle B, g(C,E,AU,\Gamma)\rangle \to \langle B', g(C,E,AS,\Gamma\setminus\{\gamma_i\})\rangle} \qquad \textit{[OR:subg-succeed]}$$

In these four rules we introduced the function $adopt(G, g)$ to define adoption of a subgoal, that is, adding the (sub)goal g to the goal base G, in order to start its achievement process. Eventually, this will result in a new belief B'.

[4]"provisional" for the reason that in these states it is not yet sure if a goal is dropped with failure or success. This depends on further achievement and failure conditions, and on an eventual process repetition (e.g. suspension or reactivation), whose formal semantics are defined later on.

The next transition rule defines how to satisfy the main precondition of the former four rules, the transition from $\langle B, adopt(G,\gamma)\rangle$ to B', that is, adopting the subgoal γ_i in order to start its achievement process, and waiting until γ_i is dropped:

$$\frac{adopt(G,\gamma_i) \rightarrow G \cup \{\gamma_i\} \qquad \langle B, G \cup \{\gamma_i\}\rangle \rightarrow \langle B', G\rangle}{\langle B, adopt(G,\gamma_i)\rangle \rightarrow \langle B', G\rangle}$$

The function $adopt(G,\gamma_i)$ adds $\{\gamma_i\}$ to the current list of adopted goals G. Finally, the new belief B' is the result of the application of transitions for the satisfaction of the goal γ_i, that concludes with some transition rule that drops γ_i from G.

Subgoals which are themselves decomposed to goals, will follow the semantics defined in this work. When they are dropped (applying *[DropSuccess]* or *[DropFailure]*, as defined later in this section) the agent's belief base is updated with $success(g)$ or $failed(g)$, where g denotes an unique identifier of a goal instance. In the case that a subgoal is a leaf goal, it will be instantiated for example according to Riemsdijk's semantics [van Riemsdijk et al., 2008]. We require that also for these goals the success or failure is annotated in the agent's belief base.

Now we define what happens if a goal is still in the state AU, but its list of subgoals Γ is empty. The following rules define when an AND-decomposed goal, which is still in AU with $\Gamma = \emptyset$ (thus, no subgoal failed), passes to the provisional success state AS. Conversely, an OR-decomposed goal fails if none of its subgoals succeeded and no SUCCESS condition is satisfied:

$$\frac{\neg\exists\langle c, \textsc{Fail}\rangle \in E.(B \models c)}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AS,\emptyset)\rangle} \qquad \textit{[AND:subg-succeed]}$$

$$\frac{\neg\exists\langle c, \textsc{Succeed}\rangle \in E.(B \models c)}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AF,\emptyset)\rangle} \qquad \textit{[OR:subg-fail]}$$

Success and failure triggered by conditions

The following rules define the possibility to transit to the states AS and AF depending on conditions related to the actions SUCCEED and FAIL. Satisfied success and failure conditions lead from AU to the states AS and AF, respectively. In the case that both conditions are true, failure conditions have precedence.

Moreover, two of these rules also consider transitions from AS to AF and vice-versa, respectively, limited to the case that $\Gamma \neq \emptyset$. The transition $AF \to AS$ will be triggered only if a subgoal of an AND-decomposed goal fails, but its achievement condition holds. Conversely, the transition $AS \to AF$ is used if in an OR-decomposed goal a subgoal succeeds, but the condition associated to the action Fail is true. By construction of this transition system, no transitions $AS \to AF$ for AND-decomposed goals, and $AF \to AS$ for OR-decomposed goals, are necessary. In the following two rules, $\mathbf{X} \in \{AU, AF\}$ and $\mathbf{Y} \in \{AU, AS\}$.

$$\frac{\Gamma \neq \emptyset \qquad \neg\exists\langle d, \textsc{Fail}\rangle \in C.(B \models d) \qquad \langle c, \textsc{Succeed}\rangle \in C \qquad B \models c}{\langle B, g(C, E, \mathbf{X}, \Gamma)\rangle \to \langle B, g(C, E, AS, \Gamma)\rangle}$$

[cond-succeedC]

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Fail}\rangle \in C \qquad B \models c}{\langle B, g(C, E, \mathbf{Y}, \Gamma)\rangle \to \langle B, g(C, E, AF, \Gamma)\rangle}$$ *[cond-failC]*

$$\frac{\neg\exists\langle d, \textsc{Fail}\rangle \in C.(B \models d) \qquad \langle c, \textsc{Succeed}\rangle \in E \qquad B \models c}{\langle B, g(C, E, AU, \emptyset)\rangle \to \langle B, g(C, E, AS, \emptyset)\rangle}$$ *[cond-succeedE]*

$$\frac{\langle c, \textsc{Fail}\rangle \in E \qquad B \models c}{\langle B, g(C, E, AU, \emptyset)\rangle \to \langle B, g(C, E, AF, \emptyset)\rangle}$$ *[cond-failE]*

Goal dropping triggered by conditions

The following transition rules define when to drop a goal from the goal base G. When dropping a goal from the state AS, the fact $success(g)$ is added to the agent's belief. Dropping it from AF, $failed(g)$ is added.

$$\frac{\Gamma \neq \emptyset \qquad g(C, E, AS, \Gamma) \in G \qquad \langle c, \textsc{DropSuccess}\rangle \in C \qquad B \models c}{\langle B, G\rangle \to \langle B \cup success(g), G \setminus \{g(C, E, AS, \Gamma)\}\rangle}$$

[drop-successC]

$$\frac{g(C, E, AS, \emptyset) \in G \qquad \langle c, \textsc{DropSuccess}\rangle \in E \qquad B \models c}{\langle B, G\rangle \to \langle B \cup success(g), G \setminus \{g(C, E, AS, \emptyset)\}\rangle}$$ *[drop-successE]*

$$\frac{\Gamma \neq \emptyset \qquad g(C,E,AF,\Gamma) \in G \qquad \langle c, \textsc{DropFailure} \rangle \in C \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup \mathit{failed}(g), G \setminus \{g(C,E,AF,\Gamma)\}\rangle} \quad \textit{[drop-failureC]}$$

$$\frac{g(C,E,AF,\emptyset) \in G \qquad \langle c, \textsc{DropFailure} \rangle \in E \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup \mathit{failed}(g), G \setminus \{g(C,E,AF,\emptyset)\}\rangle} \quad \textit{[drop-failureE]}$$

Reactivation, Suspension, and Retry

Goal achievement might not always be straight forward. Often, a failure-avoiding behaviour is desired: the agent should try alternatives or repeat the whole goal satisfaction process, to achieve goal success. Moreover, some goals will require the permanent maintenance of some state of affairs. Thus, we introduce transition rules guided by conditions and used to backtrack in the goal achievement process, either from the success or the failure state.

The following two rules define how to restart the goal achievement process, including subgoal deliberation, when in the state AF. Remaining subgoals in Γ are deleted. Such a transition is needed to repeat goal achievement if subgoal achievement failed and goal failure should be avoided. It is worth noticing that, if more transition rules are applicable at the same time and no rule is more specific than the others, precedence to the application of transition rules is given by the order of definition of the conditions at the instantiation of a goal.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Reactivate} \rangle \in C \qquad B \models c}{\langle B, g(C,E,AF,\Gamma)\rangle \rightarrow \langle B, g(C,E,AD,\emptyset)\rangle} \quad \textit{[reactivateC]}$$

$$\frac{\langle c, \textsc{Reactivate} \rangle \in E \qquad B \models c}{\langle B, g(C,E,AF,\emptyset)\rangle \rightarrow \langle B, g(C,E,AD,\emptyset)\rangle} \quad \textit{[reactivateE]}$$

Similar rules are needed for goal suspension after a successful goal execution (this is typically needed for maintain-goals). Note that, in these transitions, the list of deliberated subgoals is emptied, thus such transitions are not suitable for modelling particular *context conditions*, which should temporary suspend a goal and subsequently reactivate it, resuming from the previous state.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Suspend} \rangle \in C \qquad B \models c}{\langle B, g(C, E, AS, \Gamma) \rangle \rightarrow \langle B, g(C, E, S, \emptyset) \rangle} \qquad \textit{[suspendC]}$$

$$\frac{\langle c, \textsc{Suspend} \rangle \in E \qquad B \models c}{\langle B, g(C, E, AS, \emptyset) \rangle \rightarrow \langle B, g(C, E, S, \emptyset) \rangle} \qquad \textit{[suspendE]}$$

The last rule defines the semantics for conditions related to the action RETRY and applies only to goals with a non-empty subgoal list Γ. *[retryC]* backtracks from AS to the undefined state AU, where goal achievement can be retried with the remaining subgoals in Γ. This transition can be applied e.g. if an OR-decomposed goal succeeds, referring to the achievement of one of its subgoals, but the goal's achievement conditions are not satisfied:

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Retry} \rangle \in C \qquad B \models c}{\langle B, g(C, E, AS, \Gamma) \rangle \rightarrow \langle B, g(C, E, AU, \Gamma) \rangle}$$

5.4 Instantiation of the Abstract Architecture

The abstract architecture for non-leaf goals in goal models, with its different actions and conditions that drive and guide the goal satisfaction process, is now adapted to the behaviour needed for the various types of goals and the interplay of their achievement and failure conditions with the subgoal achievement process. We instantiate the architecture giving precise semantics for the most significant goal types: perform-goals, achieve-goals, and (reactive) maintain-goals, as introduced in Section 5.2.

5.4.1 Perform-goals

Perform-goals are available in most agent languages, to execute plans without defining some particular state to be reached [Dastani et al., 2006].

In a goal model, we associate the following semantics to a perform-goal: depending on the decomposition type, all (for AND) or at least one (for OR) of the subgoals have to be satisfied to achieve the goal. The following instance of our abstract architecture defines a simple perform-goal, for which no explicit conditions can be defined. The transitions from AU to AF and AS thus depend solely on subgoal satisfaction. The goal fails if subgoals cannot be achieved at the first try, and succeeds, otherwise.

$$P \equiv g(E, C), with\ \ E = C = \{\langle true, \textsc{Activate}\rangle,$$
$$\langle true, \textsc{DropFailure}\rangle, \langle true, \textsc{DropSuccess}\rangle\}$$

Alternative run-time semantics associated to perform goals define that failure has to be avoided and thus goal achievement has to be restarted if the goal enters in a failure state (also called *recurrent* or *retry-perform goals* in literature). This can be realised by replacing, in both E and C, the condition $\langle true, \textsc{DropFailure}\rangle$ with $\langle true, \textsc{Reactivate}\rangle$. In this interpretation, also failure conditions can be needed. Failure conditions will be detailed for achieve-goals.

5.4.2 Achieve-goals

In general, achieve-goals have a *success condition* (or *achievement condition*) s that has to be satisfied, and usually also a *failure condition* (or *drop condition*) f. Only these two conditions guide the dropping of an achieve-goal from the goal base, regardless of the satisfaction of subgoals. For example, if all subgoals fail, but the goal success condition is satisfied, then the goal is dropped with success. Moreover, we define that the success and failure conditions should be tested not only at the end, but also during subgoal achievement.

The achieve-goal can also have various behaviours to manage failure: if the success condition is still not met after the subgoals are processed, the goal can a) restart the achievement process or b) fail completely. Moreover, adding the condition $\langle \neg s, \textsc{Retry}\rangle$ to the set C of conditions tested if $\Gamma \neq \emptyset$, we can achieve the failure-preventing behaviour shown in the example in Section 5.5, that is, for failed OR-decompositions, goal achievement is restarted with the remaining subgoals. The following instantiation models the behaviour a):

$$A(s, f) \equiv g(E, C),\ with\ E = H \cup \{\langle \neg s \vee f, \textsc{Fail}\rangle\}$$
$$and\ C = H \cup \{\langle f, \textsc{Fail}\rangle, \langle \neg s, \textsc{Retry}\rangle\}$$

with the following set of conditions H, included in both E and C:

$$H = \{\langle true, \textsc{Activate}\rangle, \langle f, \textsc{DropFailure}\rangle, \langle s, \textsc{Succeed}\rangle,$$
$$\langle s, \textsc{DropSuccess}\rangle, \langle \neg s, \textsc{Reactivate}\rangle\}$$

Behaviour b) can be obtained from the previous one replacing $\langle \neg s, \textsc{Reactivate}\rangle$ with $\langle \neg s, \textsc{DropFailure}\rangle$.

5.4.3 Maintain-goals

As discussed in Section 5.2, we limit to *reactive* maintain-goals, that are endowed with a *maintenance condition* m and in most languages also with a *drop condition* d to remove the goal from the list of goals to pursue [van Riemsdijk et al., 2008].

Intuitively, maintain-goals try to maintain a certain condition true and never end their life-cycle, unless they are explicitly dropped from the set of adopted goals (that is, from the set of goals the agent actively pursues at a certain moment). The transitions correspond to the ones in achieve-goals, but the goal is suspended if m is satisfied and dropped only if d is true:

$$\begin{aligned}M(m,d) \equiv g(E,C),\ &with\ E = H \cup \{\langle \neg m \vee d, \textsc{Fail}\rangle\}\\ &and\ C = H \cup \{\langle d, \textsc{Fail}\rangle, \langle \neg m, \textsc{Retry}\rangle\},\ where\\ H = \{\langle \neg m, \textsc{Activate}\rangle, \langle d, &\textsc{DropFailure}\rangle, \langle m, \textsc{Succeed}\rangle,\\ &\langle m, \textsc{Suspend}\rangle, \langle \neg m, \textsc{Reactivate}\rangle\}\end{aligned}$$

Some definitions of maintain-goal include also a *target condition* t. Having both a maintain- and a target condition, the goal is activated each time the maintain-condition is violated, while it is suspended only if the target condition is satisfied. These property allows for a behaviour with a hysteresis in goal activation, preventing unwanted continuous switching between activation and suspension. For example, if room temperature has to be maintained at 20 °C, each time the heating is turned on, it should heat till 22 °C. To obtain such a behaviour, the goal architecture can be instantiated as $M_t(m,t,d)$, with all occurrences of m in $M(m,d)$ changed to t, except for the condition $\langle \neg m, \textsc{Activate}\rangle$.

5.5 Application of the Semantics

We illustrate the application of the proposed operational semantics, using the cleaner agent example introduced in Section 5.2.1 (Figure 5.1), Manually executing some steps of this example, we explain the expected run-time behaviour.

We detail the satisfaction process for the goal `RoomClean (RC)`, with the achievement condition that the room has to be clean (supposing the belief base will then contain the predicate *room.clean*). The goal is OR-decomposed into two goals `Wet-Cleaning (WC)` and `DryCleaning (DC)`, both goals of type *perform*, and thus without any specific achievement condition. We suppose that the cleaner agent is working in a room, but after a while it encounters some stubborn dirt that cannot be completely removed by the broom (e.g., colour spots after painting the walls). In this example we expect the behaviour outlined in the scenarios in Section 5.2.1: the cleaner first pursues `DryCleaning`, due to a higher contribution to the softgoal `efficiency`. Sweeping succeeds (plan `sweep`), but the floor is still not clean and so the agent, to avoid failure, also cleans using the mop (plan `mop`).

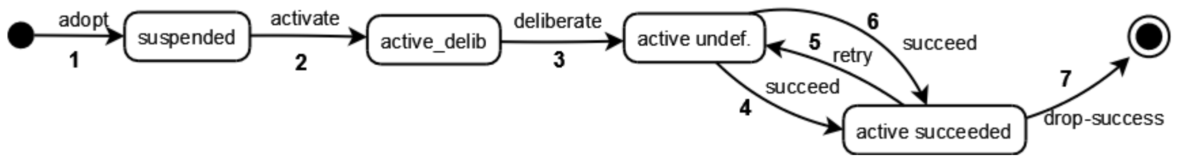


Figure 5.3: Possible life-cycle for goal `RoomClean` in the Cleaner Agent example.

We show a step-by step application of the proposed transition rules, instantiating the achieve-goal `RoomClean` with the transition rules for an OR-decomposition. We apply the set of conditions defined in Section 5.4 for achieve-goals of the form $A(s, f)$, with satisfaction condition $s{=}room.clean$ and without failure condition ($f = false$).

After the adoption of `RoomClean`, the condition $\langle true, \textsc{Activate}\rangle$ enables the application of the rule *[activateE]* (note that, at the beginning, still $\Gamma = \emptyset$) and the goal passes from state S to state AD (**2** in Figure 5.3). Then, with the rule *[deliberateE]*, the state changes to AU (**3**), and the function *deliberate* returns the subgoals $\Gamma = \{$`DC, WC`$\}$. Now we expect that `DC` is adopted and executed and returns with success. Notice that here we do not cope with the operational semantics for subgoal prioritisation, e.g. by softgoal contribution. The rule *[OR:subg-succeed]* now applies (**4**), as shown:

$$\frac{\dfrac{adopt(G, DC) \rightarrow G \cup \{DC\} \qquad \langle B, G \cup \{DC\}\rangle \rightarrow \langle B', G\rangle}{\langle B, adopt(G, DC)\rangle \rightarrow \langle B', G\rangle} \qquad B' \models success(DC)}{\langle B, g(C, E, AU, \{DC, WC\})\rangle \rightarrow \langle B', g(C, E, AS, \{WC\})\rangle}$$

The execution of `DC` can be derived by an application of transition rules that end with a rule which models *success*(`DC`) in B'. Now, to apply a transition starting from the 'provisional' success state AS, the condition s has to be evaluated. As just described, there are still some colour spots on the floor and thus s is not satisfied. Therefore, the only true precondition of a transition rule from AS is that of *[retryC]*: $\langle \neg s, \textsc{Retry}\rangle$. So the goal state changes back to AU (**5**). The only goal remaining in Γ, `WC`, will now be pursued.

We suppose that after having cleaned most of the floor, this subgoal fails, because the robot runs out of water. However, the stubborn spots were removed, and thus the condition s is satisfied. The rules *[OR:subg-fail]* and *[cond-succeedE]* are now candidates for the next transition. Since $B \models s$, only the latter can be applied and the goal state changes again to AS **(6)**. Finally, the rule *[drop-successE]* can be applied **(7)**, the goal is dropped and the predicate *success*(`RoomClean`) is added to the agent's beliefs.

On this simple example we can already observe that the agent exhibits a failure-preventing behaviour, by means of reasoning on the structure of its goal model, taking advantage of the modelled variability.

5.6 Discussion: Goal Types in Goal Models

Endowing goals in goal models with the semantics defined in this section allows designers for modelling a wide range of complex agent's behaviours by combining goal AND/OR-decomposition with different goal types and conditions. However, not all combinations are meaningful, either for modelling or for implementation purposes.

For example, performing the refinement process within a goal model, a maintain-goal can be either (a) decomposed to more specific maintain-goals, or (b) by defining the goals to achieve or perform, in order to maintain the required state. However, at run-time, maintain-goals have the property that they are not dropped when they reach the desired state, but suspended, waiting for reactivation. Thus, subgoals of the type *maintain* would never return a positive or negative outcome to their parent goal, unless they are explicitly dropped from the goal base.

For this reason, to achieve a predictable behaviour, we set as –not necessarily minimal– restrictions to goal model implementation: only the leaf-most maintain-goals should be implemented, and decomposition of achieve- and perform-goals to maintain-goals is not allowed. To condensate these guidelines into a single expression, we define g as a high-level goal, whose implementation is not desired (including maintain-goals of type (a), as defined above), and m, a and p as the three main goal types, respectively. The regular expression $g^*m?[ap]^*$ has to hold for each path from a root goal to a leaf goal in the goal model at design-time.

5.7 Related Work

Up to our knowledge, currently no agent-oriented language natively supports goal models at run-time. This entails also a lack of formalisation of the details of goal model satisfaction at run-time.

Few related work goes beyond the definition of semantics for the achievement of declarative goals by goal-plan relationships. Rule-based agent languages, which define goals by list of atomic predicates (e.g. 3APL [Hindriks et al., 1999])are able to define subgoals in the logical sense (e.g. the predicate p is a subgoal of the goal $p \wedge q$). This is a special case, covered by our semantics. However, in an open world and including the use of imperative languages, a logical correspondence between goals and subgoals cannot always be achieved.

Although coming from a different domain, *Formal Tropos* [Fuxman et al., 2001] goal models can also be executed, by simulation. However, goal decomposition semantics were not defined. Thus, consistency checking between goals in a goal hierarchy and the definition of the interplay of the different goal types in this hierarchy, e.g. by binding goal achievement to the success of one or all subgoals, is left to the engineer which does the formalisation.

We would like to mention a definition of so-called *subgoal semantics* by Riemsdijk et al. [van Riemsdijk et al., 2005], which introduces a declarative notion of subgoals within the 3APL language, in which subgoals (i.e. goals activated by a plan) are limited to a procedural behaviour. In essence, their achievement is tested after executing a plan triggered by that goal, and alternative plans are generated until the goal is achieved. This pro-active behaviour goes beyond purely procedural, event-driven semantics, as in the languages Jack and Jason[5]. Similar semantics, which include a persistent concept of goal, are however already natively included in languages such as GOAL [Hindriks et al., 2000] and Jadex and covered by our formalisation.

Sardina et al. [Sardina and Padgham, 2010] extend the BDI agent language CAN along two lines: failure handling and lookahead planning. *Subgoals* are defined as goals which have alternatives (in contrary to "motivating" (high-level) goals). At a failure, goal achievement can thus be restarted with another alternative, and blocks retrying the same plans, if no alternative is available. However, a concrete representation of the goal hierarchy, as in our approach, is missing and no "backtracking" is possible to find higher level alternatives. Moreover, the important class of maintain-goals, which

[5]Although, in Jason simple achieve-goals can be realised by a recursive definition of a goal in its plan, e.g. $!p \leftarrow xyz, !p$.

necessitate for a “suspended” state, is not representable. Second, the work adopts hierarchical task network (HTN) planning techniques within the BDI execution framework to select between possible alternatives in advance, an replanning on-line each time relevant parameters change or plans fail. This approach is complementary to ours, offering an effective mechanism for alternatives selection, an issue which our semantics do not deal with.

In available languages with a persistent goal concept, such as Jadex, GOAL, and 2APL [Dastani, 2008], which adopts ideas from GOAL, goal decompositions and alternatives can be defined only by the use of intermediate plans. GOAL and 2APL are based on formal semantics, but do not support “complex” goal types such as maintain-goals. Jadex, on the other side, implements a variety of BDI flags to customize goal processing (e.g. Retry, Repeat), but follows a pragmatic implementation approach and no attempt is made for a generic formalisation with an uniform set of operations. Our abstract architecture is able not only to handle goal decompositions, but also to define formal semantics for a big part of Jadex goal types with their BDI flags and conditions.

Starting from [van Riemsdijk et al., 2008], and referring to our approach, with a similar life-cycle, Thangarajah et al. [Thangarajah et al., 2010] cover an even higher variety of goal types, but no goal decomposition. The approach moreover promises to support proactive maintain-goals. Such goals are suspended until their maintain-condition is “predicted” to become false, and thus need a prediction mechanism, i.e. following [Duff et al., 2006]. However, as mentioned previously, with this premise proactive maintain-goals can also be handled by our approach.

5.8 Final Considerations

In this section we gave a definition for the interplay between goal models–conceived as graphs of goal AND/OR decompositions– and goal types along with their achievement conditions. To deliver on this aim, building upon a proposal by Riemsdijk et al. [van Riemsdijk et al., 2008], we characterized the behaviour of goals in goal models at run-time, providing their operational semantics.

Goal models allow designers to characterize an agents’ behaviour in terms of (less and more concrete) goals and their relationships. Representing these models at run-time and defining how they guide the run-time behaviour, an agent is able to use the information available in the models as a means for run-time adaptivity and fault tolerance.

The presented semantics currently cover only a subset of goal model concepts and relationships and do not cope with the behaviour resulting from complex reasoning mechanisms, as available for goal adoption, optimisation, conflict resolution, learning or decision making. Also, these semantics are not amenable for formal verification, e.g. by model checking.

An automatic mapping from *Tropos* goal models to Jadex agent code, the ***t2x*** tool (Section 4.3.2), adds a new layer of abstraction on the Jadex agent framework, to support whole goal models at run-time, for an automatic mapping from *Tropos* goal models to agent code. It supports the three goal types *achieve*, *perform*, and *maintain*, and maps them accordingly, to obtain a behaviour as defined in this chapter. An important future work would be the use of the defined semantics to test if the run-time behaviour of an agent is compliant with its goal-directed design.

The modelled behaviour does not only adhere to the semantics of goal AND/OR decomposition, but it is also driven by the intrinsic nature of goal types along with their achievement conditions; thus, suitable to model run-time adaptivity and failure tolerance. Concluding, we remark that a systematic analysis and formalisation of the semantics is essential, in order to be able to understand the interplay between goal model hierarchies and the detailed goal satisfaction behaviour, and to study how this could be incorporated in an agent programming language.

Chapter 6

Modelling Adaptation by Self-Organisation

6.1 Introduction

Nowadays, networked systems require decentralized and flexible configurations, which are able to support mediated services (e.g. flight booking systems) as well as peer-to-peer business/social relationships. Such software systems need to exhibit an increasing level of self-adaptivity in order to operate efficiently in a dynamically changing environment.

In the previous chapters, agent self-adaptivity has been studied at the level of an individual agent, which has the ability to perceive the surrounding environment, to interpret collected information and to reason on it. This enables the single agent to decide which behaviour to adopt in a context-aware manner.

In this chapter, we try to apply the concepts inherent to *Tropos4AS* to the engineering of multi-agent systems (MAS), targeting highly distributed systems and agent societies without centralised control, which have to exhibit properties of self-adaptivity to satisfy the goals delegated from their stakeholders in a dynamic, unknown environment. Such systems have to autonomously organize their internal structure to effect adaptivity to their changing context for achieving their global, high level goals by cooperation of the single agents.

Tropos4AS is still limited in capturing the dynamics in relationships between agents in a system, providing dependency links, which are not suitable for capturing the dynamics of interaction and interaction change at an instance level. We extend *Tropos4AS*, which follows a top-down, goal-oriented approach, with concepts from the

MAS development methodology ADELFE [1], to obtain a process for the definition of decentralised, self-organising systems.

ADELFE [Bernon et al., 2002] is a methodology tailored to the engineering of self-adaptive multi-agent societies of cooperative agents, in a bottom-up approach, by the definition of agent interactions. Self-organisation of the agents in the system leads to an emergent system self-adaptation. A MAS in ADELFE is defined from various viewpoints: the *system point of view*, which describes the system and its surroundings in terms of entities (perceptible objects); the *agent point of view*, which represents agent-internal characteristics; and the *cooperation point of view*, which represents Non-Cooperative Situations an agent is likely susceptible to encounter.

Our approach integrates ideas and concepts from ADELFE to the *Tropos4AS* methodology, to guide system decomposition to agents to reflect the entities in the domain, and to model cooperation between system agents and their reaction to situations that are non-cooperative. This is done along two lines: *1)* extending the *Tropos4AS* modelling language meta-model by including concepts from the ADELFE meta-model; and *2)* revisiting the *Tropos4AS* design process by including ADELFE activities.

The resulting design process couples *Tropos4AS top-down* analysis of the intentions of the system's stakeholders with a *bottom-up* approach to the design of interactions definition of the single agent's interactions. Thus, it allows to model decentralised, self-organizing multi-agent systems (MAS) starting from a *Tropos* requirements model, and to capture agent coordination and reactions to non-cooperative situations, enabling agents to optimize their choices, then giving rise to emergent adaptation of the global MAS.

From the viewpoint of ADELFE, the top-down goal decomposition reduces the gap from the system's objectives to the agent's activities, while as a major benefit from the viewpoint of *Tropos4AS*, the agent organisation is formed by a bottom-up definition of collaboration between agent instances. These achievements were possible also thanks to a collaboration with the IRIT research centre at the University of Toulouse, France.

The process is applied to a conference management system example, giving a first evidence for its benefits.

[1]See Section 2.4.3 for a brief introduction on ADELFE, including the concepts relevant for this chapter.

6.1.1 Comparing the two methodologies

Tropos4AS and ADELFE are founded on very different principles and have a different scope. While ADELFE is tailored to decentralised, adaptive complex systems and follows a bottom-up approach to eventually reach the global goal of the system in an emergent way through agent cooperation, *Tropos*/*Tropos4AS* claims to be a general methodology, where the system goals elicited by analysing the organisational settings, through steps of refinement and decomposition lead to program components implementable in software agents.

Albeit in both methodologies, agents are a metaphor for an autonomous entity with own goals and abilities, trying to achieve their local goals, the process of obtaining the single agent's goals presents conceptual differences. ADELFE agents are identified and their behaviour specified, analysing the domain entities, their role in the system and the relationships between them. They create a complex organisation by having at runtime a high number of instances for each agent type. The global goal of the system, which the stakeholders want to obtain from the software system, is modelled in use cases, but this global goal is not coded by the single agents and can only be observed, emerging from the collective behaviour.

The *Tropos4AS* development process starts with capturing the objectives of the stakeholders, in the requirements analysis phases. The MAS architecture is then obtained by analysing the organisational settings with the goals and tasks delegated to the system, decomposing them and delegating their satisfaction to single actors (roles or agents), following general engineering rules to achieve low coupling and high correlation between the tasks to be achieved by a single actor.

To define structure and abilities of the single agents, in ADELFE, a central role is given to agent interaction and coordination, specifying behaviour rules and associated activities both for the agent's **nominal behaviour** (i.e. the ordinary behaviour exhibited by the agent in a working situation without problems and failures) and its **cooperative behaviour** (Especially focusing on how to react to collaboration problems). Moreover, the agent's belief (called *representation*) of the outside world, and its sensors and actuators are defined.

Tropos4AS agents are characterised by the goals delegated from stakeholders and the dependencies to other agents; the nominal behaviour is defined by the goal model, including plans to perform and resources to provide to achieve goals. The goal runtime behaviour can be further specified, defining goal types and conditions on to the environment perceived by the agent. Exceptional behaviour can be defined by modelling possible failures, the errors causing them, and proper recovery activities.

In the following, we try to integrate aspects from ADELFE in *Tropos* to obtain a system which is modelled in a goal-directed way, but architected as a collective organisation of cooperative agents.

6.2 Modelling of Self-Organising MAS

6.2.1 Integration of *Tropos4AS* and ADELFE concepts

Tropos and its extension *Tropos4AS* follow a top-down approach from the system to the single agents and their behaviour, and achieve traceability by decomposition and delegation of goals through the design phases. *Tropos* requirements modelling is prominent for it's ability to capture the organisational settings where the system to develop will be integrated and the dependencies and responsibilities of the agents in the system and the actors playing different roles in the organisation. However, *Tropos*, as well as *Tropos4AS*, lack of support for agent organisations, i.e. for modelling the dynamics of collaboration between software agent instances in a multi-agent organisation where each modelled agent has various instances, which can also be dynamically added and removed.

The ADELFE methodology was created specifically for the development of such agent organisations. However, it adopts a bottom-up approach, to achieve the system's goal in an emergent way; the relationship between global goal and single agent's behaviour is not modelled and the global goal can only be observed from action and interaction of the parts.

Integrating ideas and modelling steps from ADELFE we enrich *Tropos4AS* for the modelling of agent organisations. A bottom-up addition of ADELFE cooperation rules (which fit into the concept of *Tropos4AS* failure modelling) will give to the run-time agent instances the knowledge for selection of and cooperation with their peers, and thus achieve an emergent self-organising behaviour to adapt to a changing environment.

6.2.2 Metamodel extension

We now investigate how to extend the *Tropos4AS* meta-model with concepts taken from the ADELFE meta-model, and revise the *Tropos4AS* design process including steps that belong to the ADELFE approach. To improve the modelling of the interplay of an agent with the artifacts and actors inside and outside the software system under development, we explicitly add the concept of agent's knowledge about itself and about its environment (Fig. 3.6).

ADELFE provides modelling of the agent's knowledge by characteristics (facts the agent is sure about), representations of the environment as perceived through sensors, and the agent's skills (Fig. 2.5). We integrate characteristics and representations (corresponding to the agent's belief) in the extended model. Information captured by *Skills*, *Aptitudes*, the agents *Actions* and its nominal behaviour, encoded in *Rules*, is mainly covered by the *Tropos* goal model, a main component of the *Tropos4AS* metamodel.

Understanding the interplay between the agent and its environment is of major importance to model a system's self-adaptivity. Adopting *Tropos4AS* concepts, *artifacts* represent the non-intentional entities (ADELFE *passive entities*) in- and outside the boundary of the system to develop. They provide an interface to the external world, to the users and also to other agents, through *social artifacts* such as a whiteboard or a communication channel.

The extended metamodel, shown in Fig. 6.1, defines an agent in the system (represented as a *Tropos* system actor) with its components: goal model, knowledge (i.e. the "belief base"), the system and the external environment.

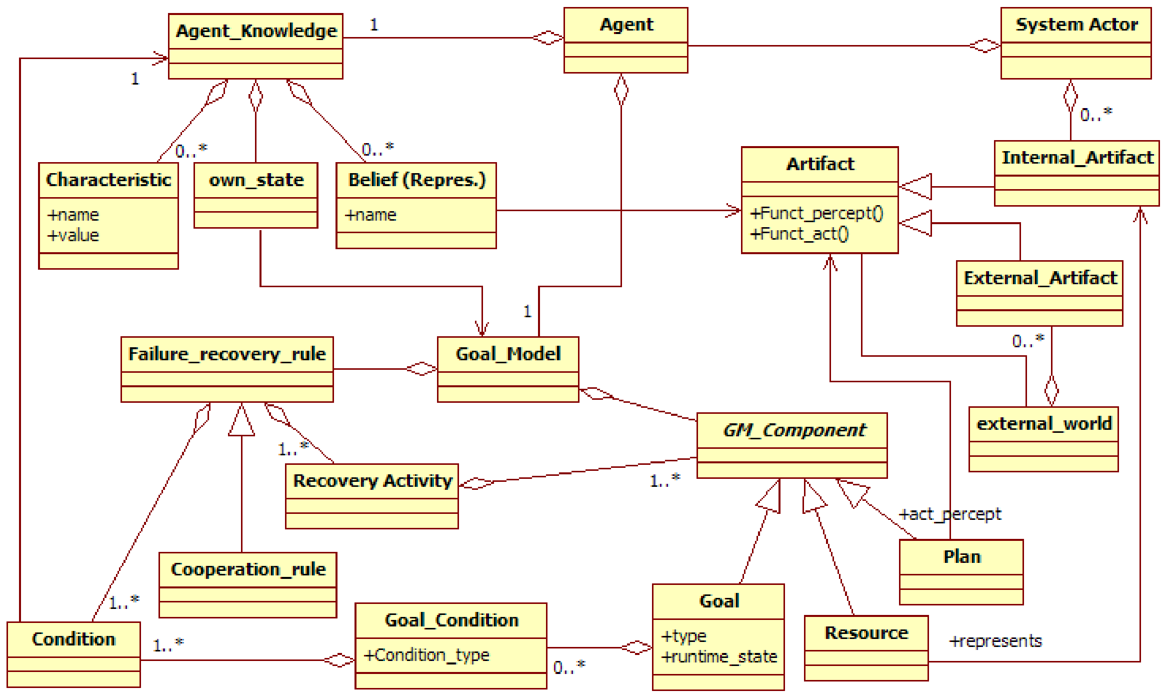


Figure 6.1: Metamodel for adaptive, cooperative agents, which extends the *Tropos4AS* meta-model with ADELFE concepts (simplified view of the *Tropos4AS* goal model).

Regarding our objective, the central ADELFE concept integrated in the methodology, is the elicitation of non-cooperative situations and modelling of the discovered so-called cooperation rules. *Tropos4AS* failure modelling is modified and concretised by allowing to directly specify failure recovery rules for goals in a goal model (class Failure_recovery_rule in Fig. 6.1). Cooperation rules are considered as a specialization of failure recovery rules, with a well defined scope. Failure recovery rules are composed by conditions on the agent's knowledge (on itself and its environment) and by recovery activities consisting of goal model fragments — a single *Tropos Plan* (which corresponds to an *Action* in ADELFE) or a more complex activity involving goals and plans.

6.2.3 Modelling Steps

We enhance the *Tropos4AS* modelling process for modelling of the newly introduced concepts. The proposed modelling steps are placed after the Late Requirements (LR) phase. As result of the LR phase, the requirements are modelled in terms of strategic dependencies between stakeholders and the software system. The system actor has its own goals, plans and resources which were derived along these dependencies. This model is given as input to the following modelling steps:

Step 1 With the LR model in input (an example is shown in Figure 6.3), define the system from the ADELFE viewpoint (i.e. activity 12 in the ADELFE methodology): identify passive and active entities in- and outside the system to develop, and identify from the active entities the autonomous agents participating in the collective task. Output: an AMAS-ML System-Environment diagram (Figure 6.4).

Step 2 In the architectural design (AD) phase, guide the decomposition of the system actor identified in the LR diagram into sub-actors (the agents in the system), according to the agents identified in the AMAS-ML system-environment diagram. The resulting *Tropos4AS* model includes agents participating to this global task, and agents achieving non-collective goals delegated by some stakeholder, or that have to supervise the collective task. The agents participating in self-organisation are annotated with a *cooperative agent* stereotype (Figure 6.5).

Step 3 With the *Tropos4AS* model resulting from *Step 2* in input, detail the high-level **nominal behaviour** of the single agents in the system by defining their goal and plan dependencies and detailing their goal models by *goal modelling* (see Section 3.3 and Figure 3.8 in this thesis), until finding the plans to achieve the goals. The environment perceived by the agent is modelled considering the passive entities identified in the previous step, and the resources modelled. From the dependencies and interactions between entities, the perception and action functionalities of the artifacts in the environment can be identified. Beliefs describe the agent's perception of these artifacts. This step is no more detailed here, as it is not central to self-organisation.

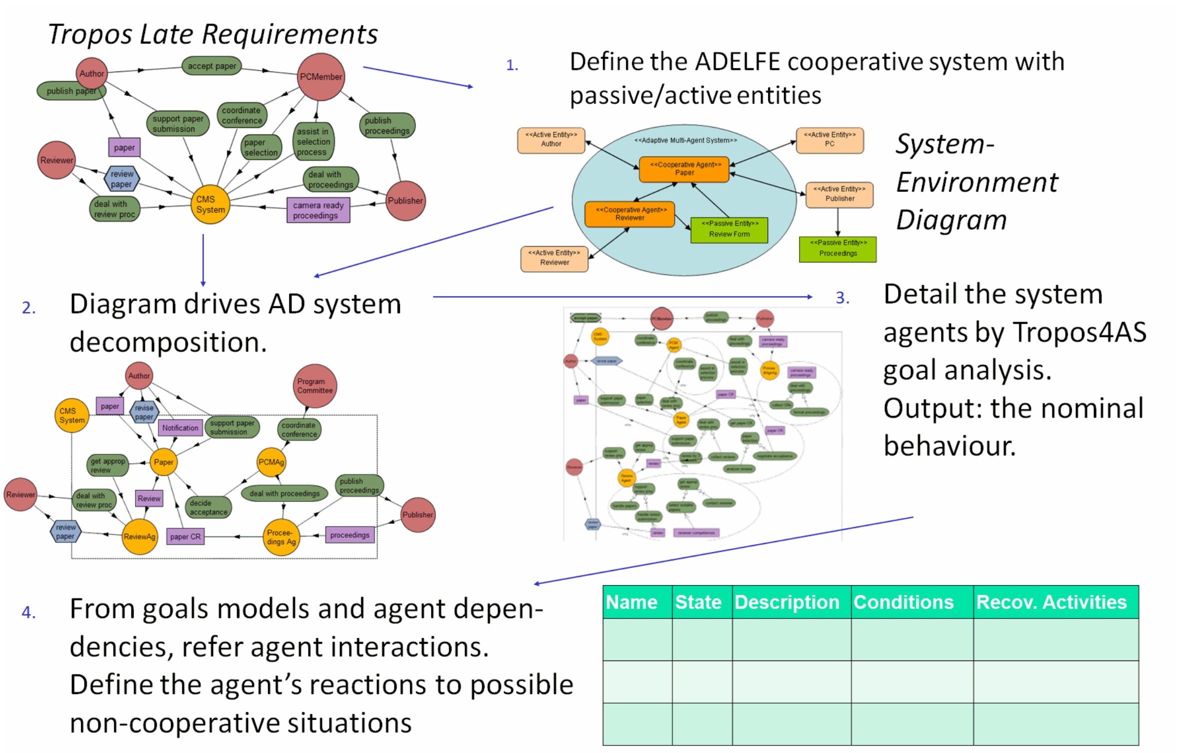
Step 4 With the *Tropos4AS* model of Step 3 in input, which includes the dependencies between agents, focus on the collective task and define the necessary interactions (i.e. activity 13 in ADELFE). Give special attention to failures that can arise from perturbations in the interaction between agents (which are cooperative by definition). The **exceptional behaviour** of each agent is now detailed by identifying non-cooperative situations that can arise. It is captured by conditions on the agent's knowledge together with the recovery activities to execute (an example in Table 6.1). These rules guide the single agent's self-organising behaviour, with activities that can be categorised in three groups: change of the own behaviour (*tuning*), change of partnership (*reorganisation*), and creation/deletion of agents (*evolution*).

An overview on the four newly introduced modelling steps is given in Figure 6.2. Next, following the *Tropos4AS* process, the goal model built in step 3 can be detailed, adding conditions, goal types and relationships, to define a more detailed nominal behaviour, and modelling possible failures not ascribed to collaboration. Modelling can continue with *Tropos* Detailed Design (DD), detailing plans (the *capability level* and low-level interactions by UML diagrams [Penserini et al., 2007b]. Following the mapping described in Chapter 4, the goal models can be mapped to *Jadex* agent code, artifacts to Java classes, and failure conditions (including cooperation rules) to goal conditions.

6.3 Application to an Example

The design process is shown on a conference management system (CMS) example, described in [DeLoach, 2002], a case study used several times for agent systems developed with different agent-oriented software engineering methodologies [DeLoach et al., 2009, Morandini et al., 2008a].

A conference management system involves several stakeholders and has to satisfy users playing various roles, such as authors, reviewers, program committee members


Tropos Late Requirements
1. Define the ADELFE cooperative system with passive/active entities
System-Environment Diagram
2. Diagram drives AD system decomposition.
3. Detail the system agents by Tropos4AS goal analysis. Output: the nominal behaviour.
4. From goals models and agent dependencies, refer agent interactions. Define the agent's reactions to possible non-cooperative situations
Author
PCMember
Reviewer
Publisher
CMS System
accept paper
publish paper
support paper submission
coordinate conference
paper
paper selection
assist in selection process
publish proceedings
review paper
deal with proceedings
deal with review proc
camera ready proceedings
<<Adaptive Multi-Agent System>>
<<Active Entity>> Author
<<Active Entity>> PC
<<Active Entity>> Publisher
<<Active Entity>> Reviewer
<<Cooperative Agent>> Paper
<<Cooperative Agent>> Reviewer
<<Passive Entity>> Review Form
<<Passive Entity>> Proceedings
Program Committee
revise paper
Notification
get approp review
Paper
PCMAg
Review
decide acceptance
ReviewAg
paper CR
Procee-dings Ag
proceedings

Name	State	Description	Conditions	Recov. Activities

Figure 6.2: Overview on the newly introduced modelling steps.

and the publisher. In the submission phase, authors need to be supported, and subsequently, $R4P$ suitable reviewers have to be found for each paper, distributing the workload evenly. For this, each paper is described by KP keywords providing its main expertise area. Each reviewer describes its expertise fields with KR keywords and should review at most $P4R$ papers.

Reviews have to be collected and evaluated to decide about acceptance or rejection of each submission, and finally the authors have to be notified, and the corrected camera ready papers collected and formatted. The prepared proceedings have then to be handed out to the publisher for printing. Figure 6.3 shows the corresponding *Tropos* LR diagram. The aim is to obtain a MAS composed by agents associated to each physical entity or role that has the need of autonomous decision and interaction, e.g. one for each paper, reviewer, etc. These agents are not "personal agents" acting selfish for the benefit of their relative stakeholder, but agents belonging to the system that are trusted and cooperative.

Interesting phases from the point of view of self-organisation between agents (which will then result to a system-mediated collaboration between physical actors or entities) are the assignment of papers to reviewers, the collection of reviews and the decision of

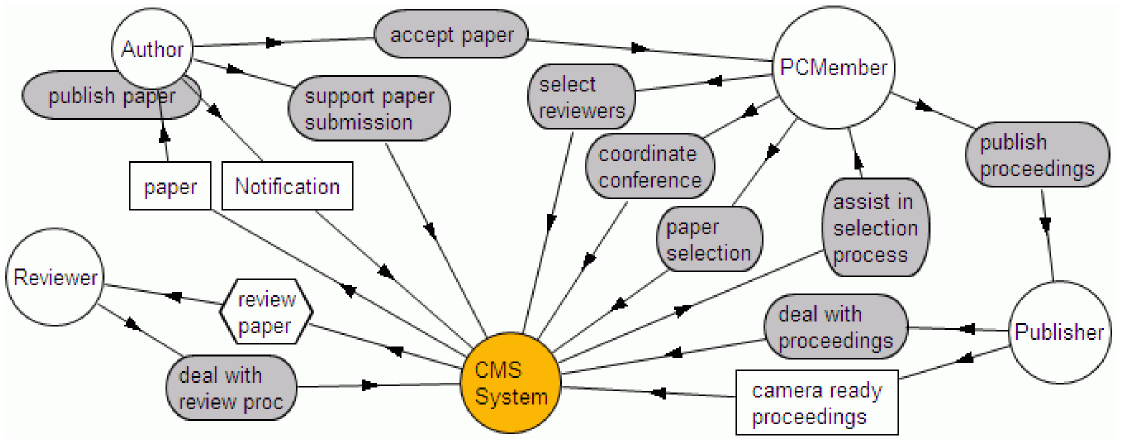


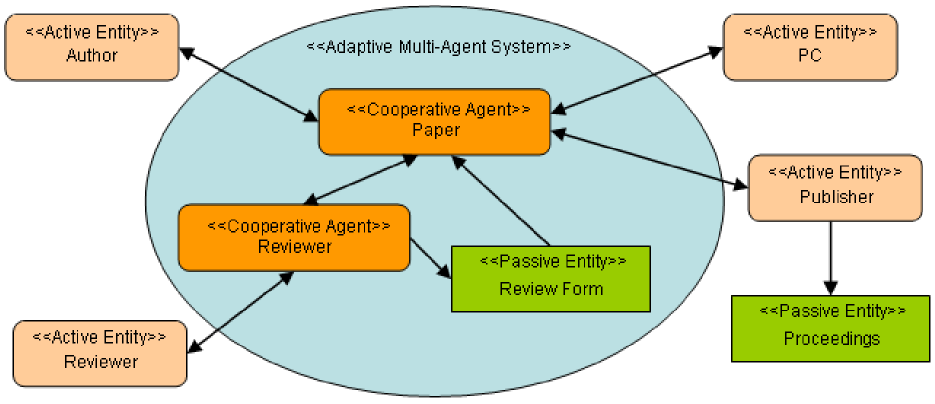
Figure 6.3: *Tropos* Late Requirements (LR) analysis: Definition of the system's objectives. Notice that dependencies between actors entail a flow of information in the opposite direction.

paper acceptance. We focus on the scenarios involving the reviewers. The reviewing process can be exposed to various kinds of perturbations. For example, unavailable reviewers, an unbalanced amount of papers in a particular area with a small number of competent reviewers, or withdrawn for any reason. Despite these eventualities could, in this small example, also be handled deterministically, they give a good example to show how a robust system should self-adapting, to meet its objectives. We now show the modelling process, going through the steps defined in Section 6.2.3.

6.3.1 Architecture

Following Step 1, we analyse the diagram in output of the *Tropos* LR phase (Figure 6.3). 6 active and 2 passive entities are identified (Figure 6.4). The active entities participating in the system's collective task are the `paper` and `reviewer agents`, representing the single submitted papers and the reviewers. We assign to the PC chair agent – identified as an agent in *Tropos*, and as an active entity (which is not participating to the collective task) in the ADELFE diagram – the charge to observe the society and to decide when a stable and optimal state is reached, in which all papers are assigned to reviewers. It will also have to advise reviewer agents to relax some constraints (e.g., allocation of more than P4R papers per reviewer).

Guided by the decomposition to agents and active entities identified in Figure 6.4, in Step 2 we decompose the `CMS system` in (Figure 6.3) into four sub-actors: `paper agent` and `reviewer agent`, which take part in the collective task of paper-review



<<Adaptive Multi-Agent System>>
<<Active Entity>> Author
<<Active Entity>> PC
<<Cooperative Agent>> Paper
<<Active Entity>> Publisher
<<Cooperative Agent>> Reviewer
<<Passive Entity>> Review Form
<<Passive Entity>> Proceedings
<<Active Entity>> Reviewer

Figure 6.4: Adelfe system-environment diagram showing the participating entities and the cooperative agents, inside the system boundary, related to the review assignment scenario.

assignment, will be associated to the single physical papers and reviewers. The `program chair` agent and the `proceedings agent` get their goals delegated from the physical actors playing the respective role in the organisation where the system is deployed and have thus also to be part of the software system (Figure 6.5).

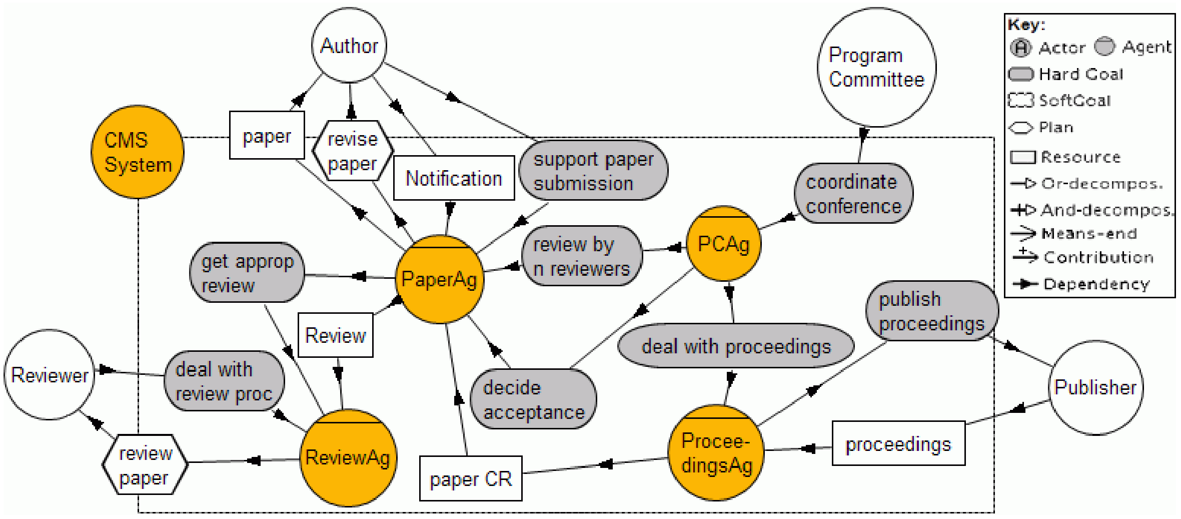


Figure 6.5: *Tropos* diagram of the multi-agent architecture.

6.3.2 Detailed design

In Step 3, the goals delegated from the stakeholders to the system are refined in the goal models of each sub-actor. Goals are decomposed until they can be operationalised by plans. Also, new dependencies between the different sub-actors arise (Figure 6.6).

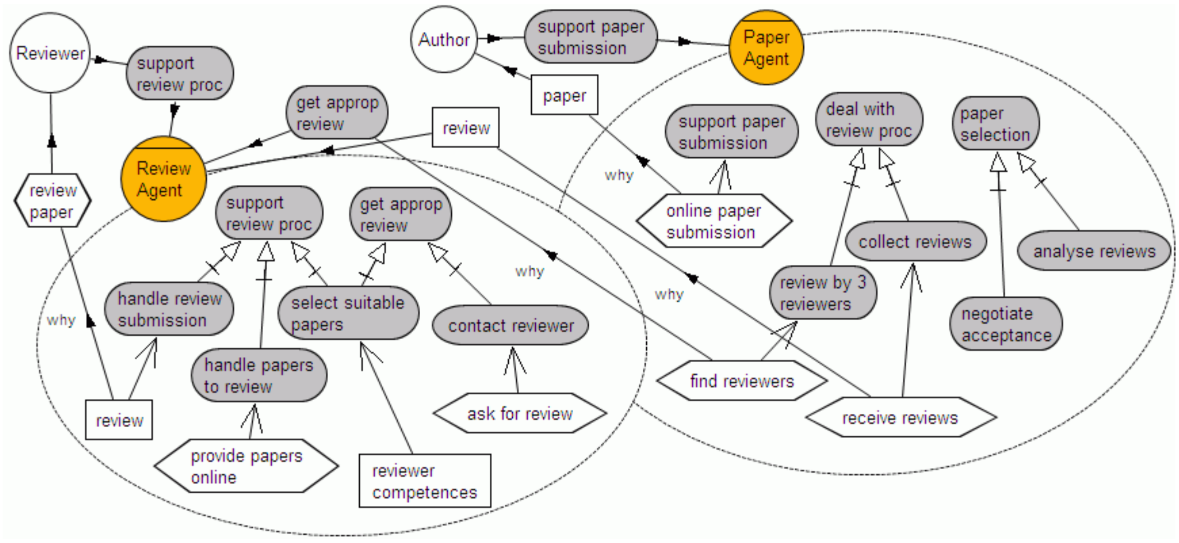


Figure 6.6: Details form the goal models of the sub-actors *Paper* and *Reviewer*.

Tropos4AS provides the means for capturing the nominal goal achievement behaviour, defining when a goal will be activated, achieved, or dropped, capturing its representation of the environment and linking its execution to environmental changes. For example, the goal `get approp review` is created after *R4P* reviewers were assigned to a paper; achieved when *R4P* reviews are collected; and failed if a review is missing at the deadline.

Agent interaction In Step 4, with the *Tropos4AS* model in output of step 3, we focus on the interactions (goal, task and resource dependencies) of the agents participating in the collective task, whose details will now be further modelled following the ADELFE process. In order to give a detailed view, we limit to the scenario of paper assignment to reviewers. *Without a centralised distribution of papers to reviewers, the relative agents have to find a relevant allocation between papers and reviewers by self-organising to achieve an optimal distribution of papers and a timely collection of appropriate reviews, being robust for possible perturbations.*

In order for papers to 'meet' reviewers, we design the system environment as a big room (a grid) where reviewers can stand on at most one square. Paper agents can move on it to find matching reviewers. This approach was already experimented with satisfactory results for a dynamic time-tabling elaboration [Picard et al., 2005]. Furthermore, we define the notion of *criticality* of a `paper agent`, a criteria to know which paper has the greatest number of constraints. It describes its difficulty to find a reviewer; it corresponds to the number of reviewers who have been met but are not relevant.

Name	**State**	**Description**	**Conditions**	**Recov. Activities**
PaperNCS2	Exploration	Two reviewers are perceived	One of them already busy	Move towards the reviewer that is free
PaperNCS3	Reviewer conflict	A paper contacted a reviewer that is already associated to $P4R$ papers	Reviewer is full	Ask the less critical paper to search for another reviewer
PaperNCS4	Highly critical	Paper agent is very critical and adequacy with reviewer $\neq 0$ (and $< KP$)	High criticality and $0<$ adequacy$<KP$	Association with reviewer is concluded
RevNCS1	No matching	No matching keywords with an arriving paper obtained	No matching keywords	Reviewer gives links to relevant neighbour agents
RevNCS2	Search promotion	Reviewer agent promotes *mutual search* by asking paper agent which reviewers were already met	No matching keywords	Remember reviewers met by paper agent

Table 6.1: Description of main NCS for Paper-agents and Reviewer-agents, activating conditions and recovery activities, which define the cooperation rules.

Nominal behaviour: Reviewer agents are placed on the grid and don't move. Paper agents are initially placed randomly on the grid and move in order to find reviewers. Each paper agent remembers the last N reviewer agents that it met, where it met them and what are the keywords associated to each of them.
Cooperative behaviour: The interactions between `paper agents` and `reviewer agents` originate from a goal dependency get_approp_review and from a resource dependency for the review. The behaviour for the cooperation between instances of these agents is defined by the agent's reaction to situations that are recognised to be "non-cooperative".

We identify and describe possible non-cooperative situations, characterised by conditions on the agent's knowledge and the recovery activities to perform (Table 6.1). In this way, the *collaboration rules* are defined, containing the activating conditions and associated recovery activities. Take the example of the paper agent: If a paper finds a reviewer that fits to its keywords but is already associated to P4R papers, the less critical of them is asked to find a new reviewer (reviewer conflict). So, if a paper agent is very critical and adequacy (keywords matching) is not null, the association with the reviewer must be established. At the reviewers side, when a paper agent arrives, adequacy is computed. If matching is not obtained, the reviewer gives hints for other reviewers in its neighbourhood which could have enough matching keywords.

To decide when to conclude self-organisation (at a point that a suitable configuration is achieved), the `PC chair` agent (which is a single instance) observes the papers, which expose their criticality and their state, ranging from satisfied to unsatisfied.

Validation The resulting design and architecture can be compared with a design of the same CMS example following the methodologies *Tropos*, Prometheus and O-MaSE, published in [DeLoach et al., 2009]. Despite it is divided into different agents, the *Tropos* architecture achieved by a top-down decomposition of the system to sub-systems is centralised and not, as defined in the original requirements, a MAS of collaborating agents. For the same example, also the Prometheus methodology provides a similar solution, while O-MaSE gives a MAS architecture similar to ours, with personal agents to support the stakeholders, but a centralised review assignment and paper selection.

6.4 Related Work

Currently, the works on methodologies focusing on self-organisation in multi-agent systems tends to increase. Tom de Wolf and Tom Holvoet [Wolf and Holvoet, 2005] proposed a full lifecycle methodology customising the Unified Process. At the requirement analysis phase, a step for the identification of high-level properties which must be shown by the running system, is added to the classical steps. The design phase is customised with two steps: one for deciding whether or not it is relevant to use a self-organising system and the other for exploiting existing practices and experiments. At the verification and testing phase, an empirical approach based on iterative development feedback is proposed. The interesting and original part of this method is that it focuses on system validation.

In [Penserini et al., 2010] the authors present a case study of a decentralised multi-agent system for ambient intelligent scenarios, motivating the need of novel organizational structures of agents that result more flexible than traditional ones, e.g. broker and matchmaker, in order to deal with context changes. The architectural design phase has been conducted by the *Tropos* modelling language in order to include the social surroundings needed to better characterize MAS architectural requirements. The resulting structure, *Implicit Organisation*, includes self-organising properties for the reassignment of the mediator role, i.e., the architectural requirement of disintermediation. Nevertheless, [Penserini et al., 2010] does not detail the agent coordination level.

Dalpiaz [Dalpiaz et al., 2010] defines interaction in a heterogeneous MAS by commitments between agents. As a possible adaptation tactic, actions are defined, which take place in case a commitment between agents is threatened. However, this adaptation is seen for the sake of the agent itself, and the effect to the MAS organisation and its emergent behaviour is not considered.

Gerhenson [Gershenson, 2005] proposes a domain-independent methodology for designing and controlling self-organizing systems. This iterative and incremental methodology includes several interrelated steps: Representation, Modelling, Simulation, Application and Evaluation. The main point of this method is that a distributed control is specified in order to influence the system (by reducing friction and promoting synergy) to ensure that it will produce the desired behaviour. This mainly philosophical work aims more at understanding these complex systems than at designing them.

Gardelli [Gardelli et al., 2008] presents an approach to engineer self-organising MAS from the early design phases. The architectural pattern adopted is based on the Agents and Artefacts metamodel [Omicini et al., 2006]. Designing a self-organising MAS consists in embedding the self-organisation mechanisms in environmental agents and properly designing their interactions with the artefacts of the environment. The design approach comprises three-steps. Modelling first provides an abstract model of the system in which user agents, artefacts and environmental agents are characterised. The second step uses stochastic simulation to study the system dynamics through statistical analysis of results, considering that proper parameters are provided for artefacts and agents. The last step consists in tuning them until the desired dynamics appear. This proposal is mainly a guide for early-design of systems based on self-organising patterns that already exist, such as natural ones.

6.5 Final Considerations

To promote the development of decentralized, collaborative MAS, in this chapter we propose to enhance *Tropos4AS* with concepts and modelling steps from ADELFE methodology. The proposed approach couples the viewpoint of a bottom-up approach to an emergent system objective with a top down analysis of the intentions of the system's stakeholders (the goal model).

The synergy of both software engineering methodologies allows to characterise a decentralised MAS by the definition of intra-agent coordination properties. The designer is now guided along a goal-oriented modelling process enhanced with specific design steps devoted to the specification of agent coordination through the modelling of recovery from non-cooperative situations. The resulting agents are able to rearrange their collaborations, leading the MAS to optimise the achievement of its current organisational goal, bringing forth an emergent behaviour. Furthermore, the conceptual modelling gains from the detailed guidelines available in ADELFE to identify system entities and agents and to define inter-agent cooperation.

If the system to develop is adequate for an AMAS approach (this can be verified, following the first steps of ADELFE), the proposed combined approach promotes the development of decentralised, distributed MAS for problem solving, and gives the possibility to deal with self-organisation of the collaboration links between agent instances, at a class (agent or role) level, which is not representable immediately in *Tropos*. The application of this approach combining the two modelling paradigms and metamodels is therefore restricted to a particular set of systems.

By *Tropos4AS* goal modelling, traceability of requirements through the design phases until the definition of the agent's behaviour is maintained, reducing the conceptual gap by maintaining the concept of goal until detailed design and – if a BDI platform is used for the implementation – even until run-time. This traceability is important especially if requirements change during system development and maintenance.

However, the link between the bottom-up approach and the objectives of the system is still not straight-forward. The emergent behaviour coming from the bottom-up approach to self-organisation, performed by modelling the single reactions to non-cooperative situations, can be only validated empirically, by observation or testing. A suitable approach for testing such MAS is proposed by Nguyen et al [Nguyen et al., 2009]. It derives *testing goals* from *Tropos* goals (e.g. for a goal `review by 3 reviewers`, a testing goal `not less than 3 reviewers`) and generates test inputs for the agents under test, by an automated, evolutionary technique. Still, we are convinced that by combination with top-down *Tropos4AS* goal analysis and decomposition, we are able to shrink the gap between global system goals and cooperation rules.

Future work concerns carrying on the implementation of systems developed with the present approach, their observation and testing, to gain experience on how to bridge the still existing gap between low-level goals and the behaviour emerging from cooperation.

Chapter 7

Evaluation Through Examples

In this chapter we present case studies, in which the *Tropos4AS* framework was applied to the development of small, illustrative systems. These resulting examples show the applicability of the framework and were used for the improvement and consolidation of the modelling language, the process and the tool support. Moreover, these examples anticipate an iterative approach, where feedback from run-time is used for an improvement of the models.

In Section 7.1, the whole *Tropos4AS* modelling process is applied to a simple cleaner robot system. A special focus is given to the novel extensions introduced with *Tropos4AS* and the forms of self-adaptivity resulting from a direct, automated mapping of the *Tropos4AS* models to an agent-oriented prototype implementation.

Section 7.2 presents a case study for a second, different cleaner robot, with the main focus on the implementation of the software running in a virtual environment, the testing and evaluation of the exhibited behaviour. In several (documented) iterations, the behaviour of the software and its self-adaptivity, are improved, considering feedback from the testing activities to improve requirements, design and implementation.

In Section 7.3 we give a first sketch for an evaluation of the behaviour of generated early prototypes, using the feedback for a refinement of the requirements. A simplified travel planner [Morandini et al., 2008e] and a on-line computer recommender system (with feedback from real users) [Tomasi, 2009] are used for a feasibility study.

Moreover, in Chapter 8, we report on a detailed empirical evaluation of the *Tropos4AS* modelling language in comparison to standard *Tropos* modelling. This evaluation consists of two experiments carried out with various subjects, to collect statistical evidence on the effectiveness (modelling effort, model correctness, model comprehensibility) of *Tropos4AS* models.

7.1 Process application to an Example

In this section, we apply the *Tropos4AS* framework for the development of a simple cleaner agent system, to illustrate the process, the models and the obtained run-time behaviour of a generated prototype.

7.1.1 Description of the system

As example to apply the *Tropos4AS* modelling process, we refer to a cleaner agent, an example for a self-adaptive system, which is used in several variations in artificial intelligence and multi-agent systems fields. This robot should be able to autonomously clean the floor where it is situated, by recognizing the different types of dirt and removing them with proper tools. To clean optimally, the robot shall autonomously choose its mechanical configuration, including cleaning tools, and the appropriate behaviour, ensuring proper cleaning for various environments (outdoor, indoor, etc.).

The CleanerSystem represents the control software for this autonomous robot, which could ideally be employed in a company building. Its main goal is to clean the company's office building, which includes offices, lavatories, as well as the outside areas. The robot can move in all directions and is equipped with two different sensors, with a mop, a broom, a leaf blower and a dust box. One of the robot's sensors is able to reveal the floor type, which can be either outside (grass, tarmac, gravel) or inside a building (linoleum, tiles). The second sensor reveals type and amount of dirt (dust, liquid, gravel, leaves or mud). To adapt to the different working environments, the robot will be able to choose among various settings of its equipment. The area to clean is plane and it is provided with automatic sliding doors, dustbins and at least one battery charging station. The cleaner has to properly clean every surface it encounters, while avoiding failure, caused e.g. by a complete discharge of its battery.

Cleaning should be carried out autonomously, including battery loading and emptying of the robot's internal dust box. The CleanerSystem should optimise the cleaning behaviour depending on the current location and the type of dirt the robot has to clean. In particular, the adjective "clean" in the requirements does not denote a particular, well defined state – cleaning has to be carried out appropriately. For example, in the outside areas, the cleaner should be configured for cleaning from coarse dirt and gravel, ignoring remaining fine dust or liquid. Conversely, in the lavatory, the robot will possibly encounter dust and liquid, and it has to configure itself (reduce trim height, slow down movement, etc.) to be able to accurately clean this type of dirt, cleaning not only the dirty spots, but sanitising the whole area. Moreover, it should be able to clean in a satisfactory way unexpected dirt types in unexpected locations, such as coarse gravel in the office (e.g. suppose that bricklayers carried out some small work), by adapting its configuration accordingly.

The CleanerSystem shall be implemented as a (simple) self-adaptive system, having alternative ways to reach its goals, being able to adapt its behaviour to the dynamic environment autonomously switching between them with a limited switching overhead, and be able to avoid basic cleaning failures.

7.1.2 Tropos modelling

We start modelling the CleanerSystem performing requirements analysis and architectural design, following the *Tropos* methodology. After the *Tropos* **Early Requirements Analysis**, where the system is modelled as-it-is (i.e. before introducing the cleaning robot), in the **Late Requirements Analysis** (LR) phase we obtain an actor diagram showing the expectations from the newly introduced system, modelled by delegation of goals and softgoals from the stakeholders (here, Director and Employee) to the system actor, as illustrated in Figure 7.1. The only hard goal delegated to the simple cleaner system is CleanEnvironment. In the system's goal model it is decomposed to two more concrete goals: MaintainPlaceClean and MaintainBatteryLoaded. For the sake of conciseness, we skip details about battery charging and dust box emptying activities.

In the **Architectural Design** (AD) phase, still following *Tropos* as described in [Penserini et al., 2007b], with actor modelling and capability modelling, the system's goals are delegated to the actors and roles that will take part of the MAS realising the system's goals. To put the focus on the modelling of a single agent, the small example has a single main system actor, the CleanerSystem. Therefore, these AD actor dependencies, displayed in Figure 7.1, are similar to the ones in output of the LR phase. We have an agent CleanerSystem, which has to achieve the main goals delegated to the system in the LR phase. The only actor added is a Movement role, to which all subgoals related to robot movement were delegated. The Movement actor will not be detailed in the following.

The goals of the CleanerSystem are now analysed and decomposed with high variability modelling in mind. To detail the goal MaintainPlaceClean, the designer defines three main alternative subgoals, represented by the goals OutsideClean, OfficesClean, and LavatoryClean. These goals are further detailed until delegating parts to the Movement role or operationalising them with plans (some of them shown in Figure 7.1).

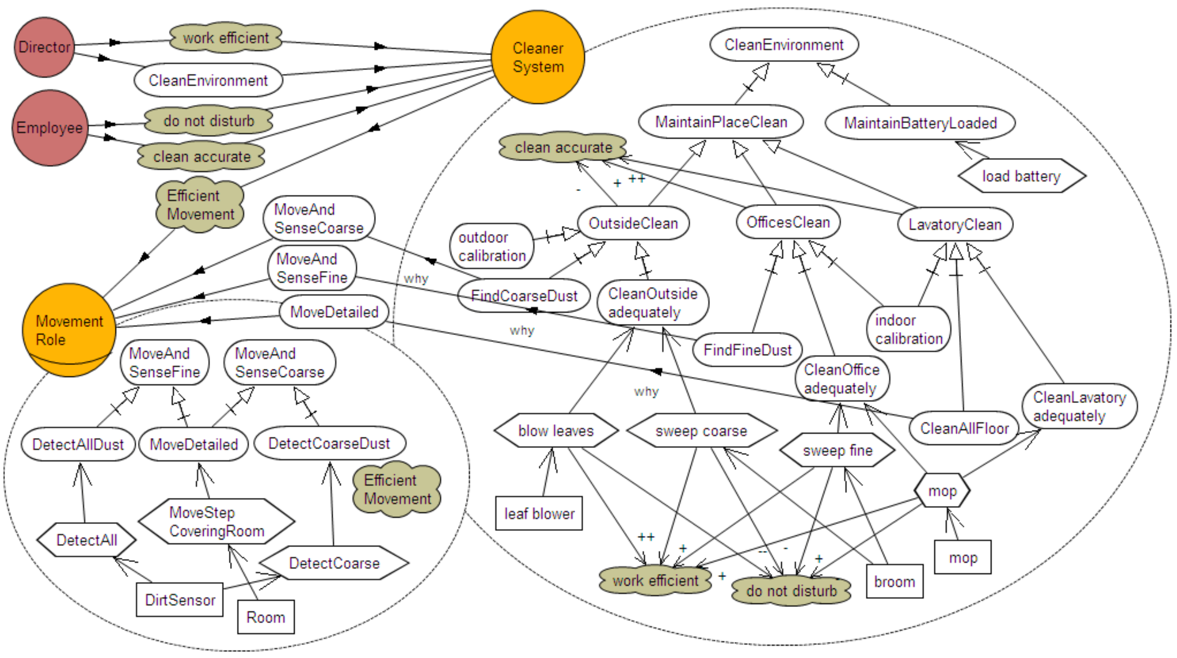


Figure 7.1: Architectural Design goal model for the CleanerSystem, obtained following the *Tropos* methodology.

As a quality criterion e.g. the softgoal clean accurate is modelled. It obtains positive contributions from OfficeClean (+) and LavatoryClean (++), while OutsideClean contributes negatively (–).

Therefore, softgoal contributions may be used as criteria for an optimization of the selection of alternatives, either at design time (following 'traditional' *Tropos*), or at run-time (the main idea behind *Tropos4AS*). Plans defining the low-level activities may be further detailed only later in the *Tropos* Detailed Design phase, which is out of the scope of this work.

The *Tropos* model in Figure 7.1 defines the *knowledge level* of the CleanerSystem agent. It shows the requirements the system has to achieve, including various alternatives to reach the root goal MaintainPlaceClean, but it is still missing important details that are of high importance to obtain a system that is self-adaptive: the relationship between the agent's goals and the environment. We bring such relationship to the knowledge level, modelling "why" an agent has to play some behaviour. As an example, the agent has the goal MaintainBatteryLoaded, to load the battery when it is charged less than 20%. However, the related dependency is not explicit, until linking the goal to the battery artifact. With the *Tropos* LR and AD models in input, now the *Tropos4AS* process is applied as described in Section 3.3.

7.1.3 Extended goal modelling

The *Tropos* model captures the goals of an agent and it's dependencies to others, but lacks a specification of the details of goal achievement, For the selection of alternatives, which will no more be done solely by the engineer at design time, but also autonomously by the agent at run-time, evaluation criteria are particularly important.

For example, the CleanerSystem modelled in Figure 7.1 has three alternatives to achieve the goal MaintainPlaceClean. However, in standard *Tropos* the engineer has no possibility to express details on what circumstances are crucial for selecting between these alternatives. By which criteria are alternatives selected (e.g. by floor type, by dirt type, or even battery level), and when will they be achieved? We give to the CleanerSystem a representation of its perceivable environment, then we explicitly relate the goals to it.

Environment modelling First, we model the agent's environment (step E1). The resources in Figure 7.1 are modelled as artifacts in the system. Moreover, the designer adds an engine in charge for the robot's movement (the MovementRole), and the sensors necessary for sensing floor and dirt type (dust, gravel,. . .). As external entities involved in the system, the area to clean, the dirt, floor and obstacles are identified to be necessary for the robot to work. The CleanerSystem will only have a partial vision of the real environment, covering the data sensed so far. Thus, for example, there will be an artifact representing the area already visited. The obtained environment diagram is shown in Figure 7.2.

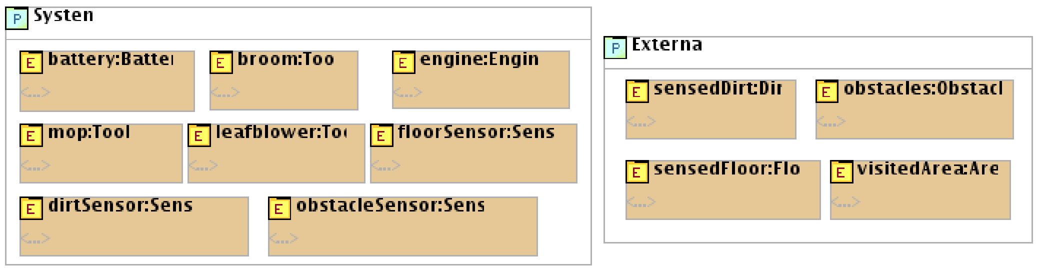
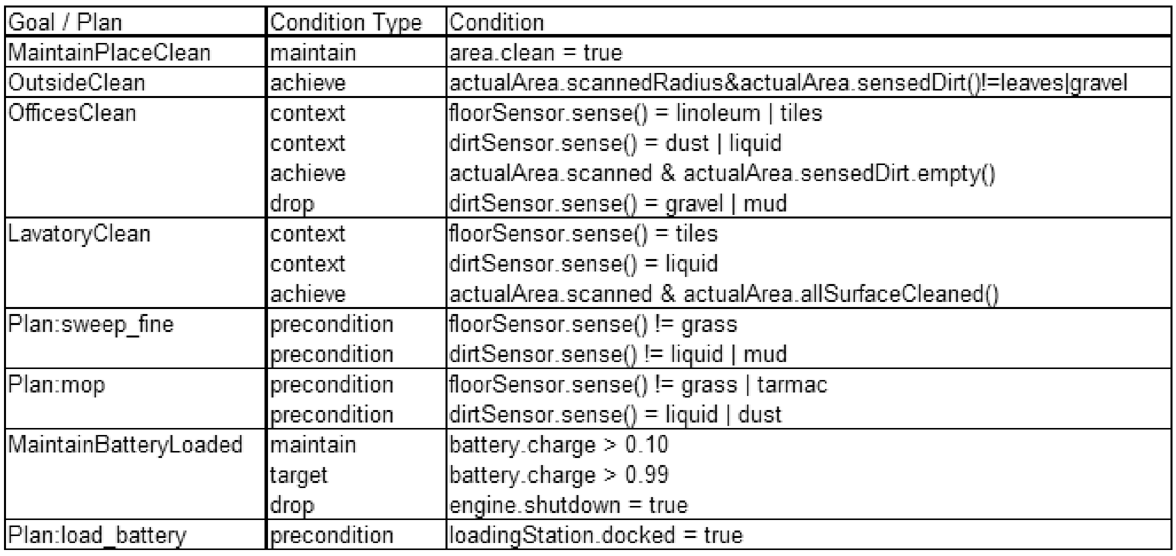


Figure 7.2: Environment model for the CleanerSystem.

Conditions modelling In step E2, the designer tries to link the achievement of goals in the system-to-be to perceptions from the environment. The modelled root goal CleanEnvironment describes a high-level goal, which is difficult to define, which is also a reason for having decomposed it, making the requirements more concrete. Thus, we decided to start condition modelling only on its subgoals.

For the goal MaintainBatteryLoaded, we defined two conditions on the battery state (also see Figure 4.12): its charge should be maintained above 10% and when loading, the battery should be loaded above 99% (target condition).

Special interest has been given to the three alternatives defined for the goal MaintainPlaceClean: adaptability to these configurations is restricted by setting context-conditions, such as 'linoleum|tiles' for the floor type sensed and 'fine|dust' for the dirt sensed, both associated to the goal OfficesClean. Moreover, such goals need an achievement condition, to determine precisely if one of the alternatives was achieved or if it would be necessary to try to achieve another one of them. For OfficesClean, we want that the whole room was visited and all the dirt removed, setting the achievement condition 'actualArea.scanned & actualArea.sensedDirt=empty'. Similar context and achievement conditions are set for lavatoryClean and OutsideClean.



Goal / Plan	Condition Type	Condition
MaintainPlaceClean	maintain	area.clean = true
OutsideClean	achieve	actualArea.scannedRadius&actualArea.sensedDirt()!=leaves|gravel
OfficesClean	context	floorSensor.sense() = linoleum | tiles
	context	dirtSensor.sense() = dust | liquid
	achieve	actualArea.scanned & actualArea.sensedDirt.empty()
	drop	dirtSensor.sense() = gravel | mud
LavatoryClean	context	floorSensor.sense() = tiles
	context	dirtSensor.sense() = liquid
	achieve	actualArea.scanned & actualArea.allSurfaceCleaned()
Plan:sweep_fine	precondition	floorSensor.sense() != grass
	precondition	dirtSensor.sense() != liquid | mud
Plan:mop	precondition	floorSensor.sense() != grass | tarmac
	precondition	dirtSensor.sense() = liquid | dust
MaintainBatteryLoaded	maintain	battery.charge > 0.10
	target	battery.charge > 0.99
	drop	engine.shutdown = true
Plan:load_battery	precondition	loadingStation.docked = true

Figure 7.3: Table defining part of the goal conditions for the CleanerSystem.

In addition, pre-conditions have been set to restrict plans applicability. For the use of the plan mop the pre-condition has been specified that sensed dirt can only be of two types: *liquid* or *dust*. On the contrary, sweeping should only be performed when the floor type is not 'grass' and the dirt sensor senses no 'liquid' or 'mud'. The table in Figure 7.3 summarizes some of the conditions set for the CleanerSystem agent.

Detailed goal modelling In Step E3 the designer defines the type of each of the goals to implement. We decide not to detail the goal CleanEnvironment, which, as already said, describes an abstract concept, concretized by decomposition in the requirements modelling phase. We start with detailing its subgoals MaintainPlaceClean and MaintainBatteryLoaded, which are straightforward defined as maintain goals. When battery loading is necessary, this goal must have precedence on cleaning. Thus, an additional «inhibits» relationship between the two goals is defined. Since the subgoals of MaintainPlaceClean will try to achieve some state in a particular environment, they are defined to be achieve-goals. The subgoals of them are either achieve or perform-goals, depending on if they have to reach some state, e.g. CleanOutsideAdequately, or to perform some action, e.g. FindFineDust, which should not fail if no dust is found. All types are defined in Figure 7.4.

The process is iterated, reworking the conditions defined in step E2, and detailing them to comply to the goal types defined. For example, a drop-condition "dirt sensor senses gravel or mud" is defined for OfficesClean. As defined in the metamodel in Figure 3.3, for achieve-goals we have to define failure or achievement conditions to drop a goal from the goal base. Thus, we change this condition's type to a failure condition.

7.1.4 Failure modelling

The nominal behaviour of the CleanerSystem has been modelled in the previous steps and would enable the agent to perform it's work in the predefined environment. With this step, we try to identify possible failures of the system (F1), to elicit missing requirements that often originate from environmental conditions that were not foreseen. In this way, we complete the model of the system, giving it the capabilities to reach it's goals also in some unexpected situations.

For the purpose of the example, we look at a possible failure of the high-level goal MaintainPlaceClean, which we name unable to clean. In step F2, we identify possible errors, such as mud detected (an error which can easily be predicted and prevented, adding new capabilities to the agent), cleaning tool broken (a non predictable, but recoverable error), and sudden power failure (which cannot be predicted nor recovered from, unless the error is prevented by additional hardware, e.g. an emergency battery circuit).

As next step (F3), recovery activities are defined: if mud is detected, the agent could cope with this error e.g. by notifying the user and return to another place to work or by using new capabilities, combining cleaning tools for a new cleaning strategy. This is modelled by the plan cleanMudWithMop, which combines two new plans mop slowly and rinse out frequently to recover from the error, using available cleaning tools. In step F4 we have to decide if to integrate the recovery activities identified for an error into the nominal behaviour of the agent, that is, its core goal model, or, if we leave them as exceptional behaviour, in the failure model. The plan cleanMudWithMop might be added in the core goal model as a means to achieve the goal CleanOutsideAdequately, with a precondition that corresponds to the detection of the error, "dirtSensor.sense() = mud". However, we suppose that the error will be a rare event and leave it as an exceptional behaviour, whose creation condition corresponds to the detection of the error. In other words, the designer increases the agent's effectiveness, putting apart this exceptional behaviour from the core part of the goal model, to get a clear separation of concerns. The resulting failure model can be seen in Figure 7.4.

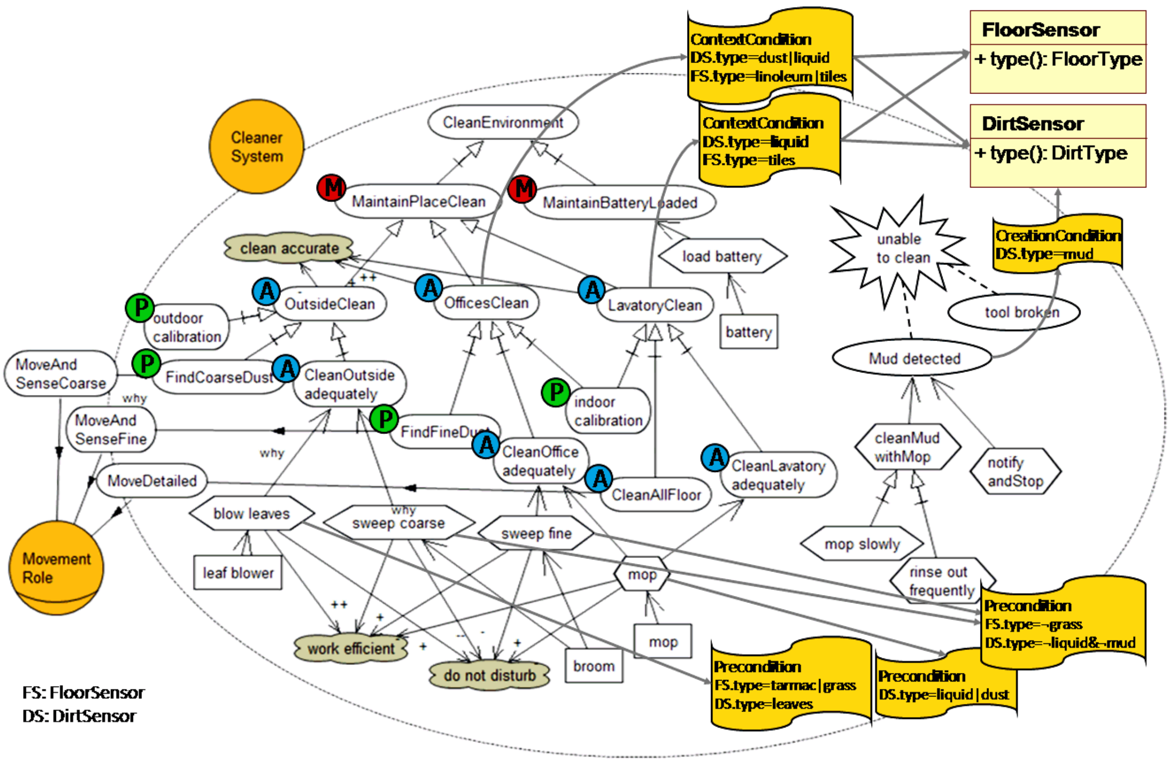


Figure 7.4: The CleanerSystem, built following the *Tropos4AS* process with its modelling extensions (some environmental artifacts are not explicitly represented for clarity reasons), with a failure model for avoiding cleaning failure.

7.1.5 Implementation and behaviour of the prototype

The models of the CleanerSystem obtained from the previous step are now mapped to BDI agent code, as defined in Chapter 4. The tool-supported code generation results in a Jadex Agent Definition File with ca. 850 lines of XML code, containing 16 goals, 30 plans (including auxiliary goals and plans), meta-reasoning structures for alternatives selection, the representation of the goal model in the belief base, and a definition of accepted messages (e.g. for communication with the agent in charge for managing the movement).

Since we are interested to the adaptive behaviour of the obtained prototype at a knowledge level, the code was modified and completed only in few parts, and implementing only the parts for the capabilities (i.e. the Java code for the plans) which are necessary for making the prototype work correctly at the knowledge level. The environment, sensor inputs and failures were directly implemented for the purpose of running various scenarios.

We show the behaviour of the CleanerSystem on a simple scenario of adaptation: The robot is achieving the goal OfficeClean, because the sensors report a typical floor profile. Suddenly, the agent senses some gravel (e.g. imagining that bricklayers carried out some small work). In this situation, we expect that the robot switches to the configuration OutsideClean until the room is clean from gravel, and then comes back to the achievement of OfficeClean for a more accurate cleaning.

Due to the modelled context-conditions, the subgoal OfficeClean is no more suitable when coarse gravel is sensed. Therefore, the system adapts by tracking back to the parent goal MaintainPlaceClean, and switches to the only remaining applicable goal OutsideClean and tries to clean the floor from the gravel. Once the gravel is removed, OutsideClean succeeded. Notice that, in the traditional *Tropos* OR-decomposition, achievement of OutsideClean would lead to the achievement of the top level goal MaintainPlaceClean. On the contrary, in *Tropos4AS*, MaintainPlaceClean has its own maintain-condition (area.clean=true), which is not yet satisfied. Since the gravel was removed, the subgoal OfficeClean is again applicable and can now be reactivated to clean the location properly from the remaining fine dust.

Selection between alternative capabilities – i.e., for the subgoal CleanOffice_adequately, the plans mop and sweep fine – is made by optimizing softgoal contribution [Penserini et al., 2007a], since both plan pre-conditions are true. In our case, the plan sweep fine is executed and the goals are achieved.

7.1.6 Final Considerations

In this section we applied the *Tropos4AS* process and models to the development of a small system, to capture various forms of variability and to show the resulting requirements-level adaptivity to changes in the environment. The main parts of the prototype, which characterise its knowledge level behaviour, were generated by automated mapping tools, and its execution shows the expected goal-directed behaviour, reasoning on the goal model at run-time. The selection of alternatives is carried out observing context conditions and softgoal contributions, while the goal satisfaction behaviour for goals in the goal model depends on the achievement (or failure) of their subgoals, as well as on their own achievement or maintenance conditions.

7.2 Development and Evolution of a Prototype

In this section we describe a case study with a second cleaner robot, which however differs from the one in the example in Section 7.1 form the requirements analysis phase on. We detail the modelling, mapping and implementation of the system, as well as various documented process iterations which take into account the feedback from an automated testing of implemented prototypes at run-time, for an improvement of the system and its adaptivity properties.

Carrying out this case study, after the requirements analysis and an accurate definition of the goal satisfaction dynamics (with goal types and conditions) with *Tropos4AS*, following the steps defined in Section 3.3, the main effort is put on the implementation and testing for giving feedback for further process iterations. The software is implemented as a BDI agent on the Jadex platform. The knowledge level definition of the agent (its goals, available plans, beliefs, etc.) in a Jadex *Agent Definition File* and the main skeleton of the Java code, were generated by the ***t2x*** tool. Capability modelling (as e.g. in [Penserini et al., 2006c]) was not performed, thus the single functionalities (represented by *plans*) have to be implemented manually. The software agent is deployed in a virtual environment. It is then tested for carrying out the desired behaviour, applying the goal-oriented testing method described in [Nguyen et al., 2010]. Feedback from this testing is given back to the design and implementation, to improve the robot's behaviour.

We show various iterations of the development process, in which bugs are corrected, the goal model is modified and functionalities added, to achieve the desired adaptive behaviour. Adaptivity features were explicitly considered for the application. However, using the ***t2x*** tool, paying attention to the correspondence between models and implementation, and with the aim of using as much as possible the features provided by the ***t2x*** implementation and the Jadex platform, only basic adaptivity features have been implemented. Further details on this case study and on the iterative modelling process can be found in the technical report [Qureshi et al., 2010b].

7.2.1 The case study: iCleaner

We have the aim to develop a cleaning robot, called **iCleaner**, which has to work autonomously to properly clean the environments (e.g. a room) assigned to it. To achieve this goal, the agent has to adapt its cleaning strategies to each environment, to maintain its battery level, to be efficient and robust. The cleaning agent needs to be adaptive to deal with the open environments where attributes of objects may change (e.g. locations of obstacles), or unknown objects may appear. Finally, it should to perform the following tasks autonomously:

1. Explore the area for important objects (waste and obstacles).
2. Collect waste and bring it to the closest bin which is not full.
3. Maintain the battery charged, by sufficient re-charging.
4. Avoid obstacles, by changing course when necessary.

The cleaning robot needs basic adaptivity features to deal with the dynamic environments where attributes of objects may change (e.g. locations of obstacles), or unknown objects may appear. Adaptivity allows the agent keep or improve its performance as well as its robustness. The performance of the agent can be calculated based on its efficiency (the waste collected in a time slot) and its power consumption, while the robustness of the **iCleaner** can be estimated e.g. by the number of crashes during a unit of operation time. Since the environment is open and dynamic, it influences the performance and robustness of the agent.

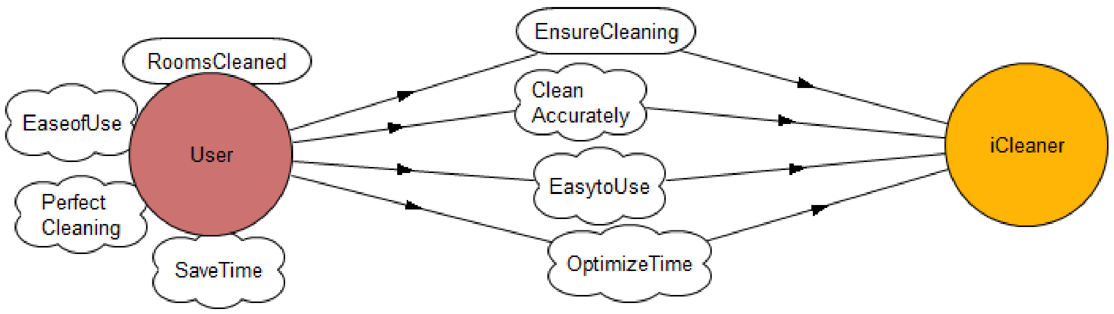


Figure 7.5: Actor diagram showing a stakeholder (User) delegating goals and softgoals to the newly introduced **iCleaner** system.

7.2.2 Applying *Tropos4AS*

Applying the *Tropos4AS* process, in the requirements analysis phases, the requirements of the stakeholder (in our case, the user of the cleaner robot) are elicited (Figure 7.5) and the goal model of the system is built, starting from the goals delegated to it, decomposing and analysing them and operationalising them with plans.

This goal model builds the basis to be exploited and extended in the subsequent iterations of the modelling process, iterating first between requirement-time and design-time and, once code is available, also through run-time and testing. The goal model corresponding to version 1 of the **iCleaner**, depicted in Figure 7.6, is used for the following illustration.

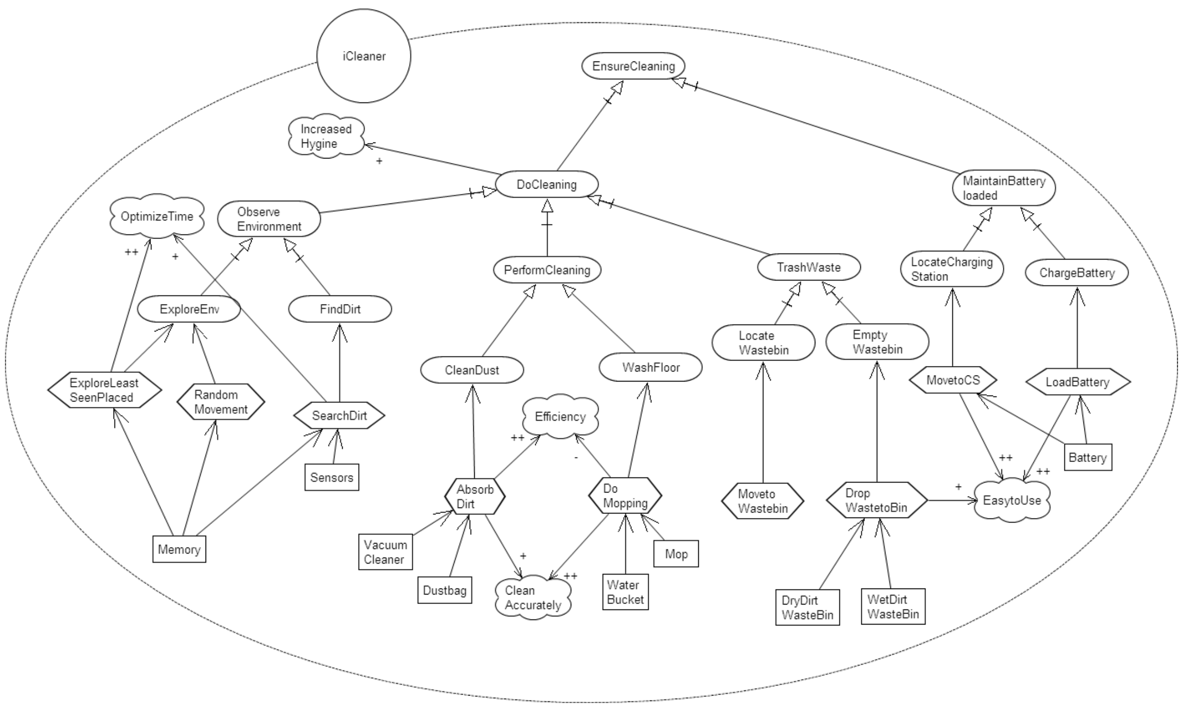
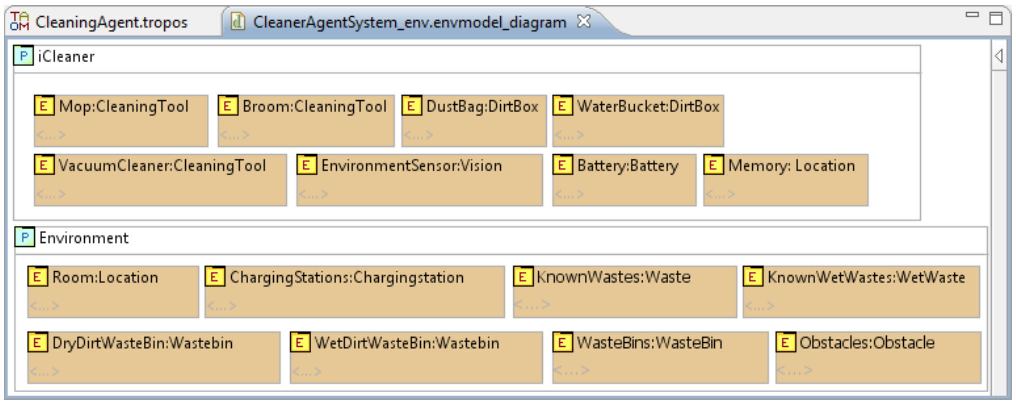


Figure 7.6: Goal model for the **iCleaner** agent, first version (V1).

In the architectural design phase, due to the simple system, no decomposition to sub-actors is performed. Now, the *Tropos4AS* process extensions, as defined in Chapter 3, are applied. In this example, we limit to extended goal modelling, which is tool-supported, and do not consider failure modelling (considering that failure models are finally also mapped to goal and plan constructs, enabled by conditions). The goal model is detailed adding an environment model, goal types, various conditions for defining goal satisfaction (e.g., achievement, maintenance) and conditions for acquiring and for dropping goals in relation to the environment.

The artificial simulation environment for the iCleaner was adopted from the testing environment used in [Nguyen et al., 2010], which extends an existing Jadex example. A domain ontology is provided which defines the interface between an agent and the environment. It defines concepts (objects) and actions available for an agent to act and percept in this environment, and is also used for the ontology-based test generation.

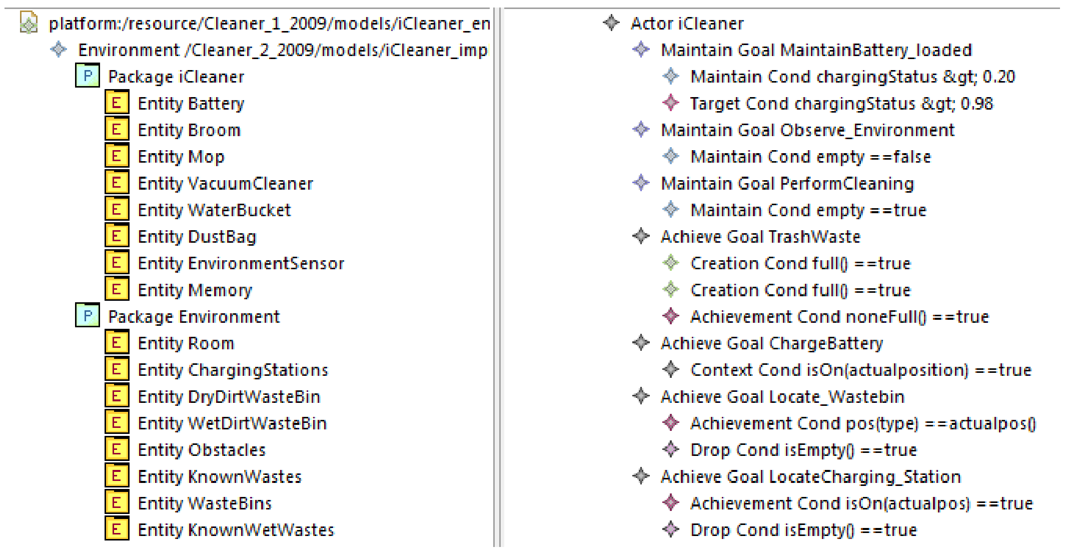
The environment model (Figure 7.7) provides the agent the perception about the domain, based on the entities specified in the domain ontology. It thus represents an instance of this ontology for all external perceptions of the system. Artifacts internal to the system, such as the battery and the water bucket were already represented as resources in the goal models at the late requirements analysis phase.


CleaningAgent.tropos | CleanerAgentSystem_env.envmodel_diagram

P iCleaner
- E Mop:CleaningTool
- E Broom:CleaningTool
- E DustBag:DirtBox
- E WaterBucket:DirtBox
- E VacuumCleaner:CleaningTool
- E EnvironmentSensor:Vision
- E Battery:Battery
- E Memory: Location

P Environment
- E Room:Location
- E ChargingStations:Chargingstation
- E KnownWastes:Waste
- E KnownWetWastes:WetWaste
- E DryDirtWasteBin:Wastebin
- E WetDirtWasteBin:Wastebin
- E WasteBins:WasteBin
- E Obstacles:Obstacle

Figure 7.7: Environment model for the iCleaner.


```
platform:/resource/Cleaner_1_2009/models/iCleaner_en
  Environment /Cleaner_2_2009/models/iCleaner_imp
    Package iCleaner
      Entity Battery
      Entity Broom
      Entity Mop
      Entity VacuumCleaner
      Entity WaterBucket
      Entity DustBag
      Entity EnvironmentSensor
      Entity Memory
    Package Environment
      Entity Room
      Entity ChargingStations
      Entity DryDirtWasteBin
      Entity WetDirtWasteBin
      Entity Obstacles
      Entity KnownWastes
      Entity WasteBins
      Entity KnownWetWastes
```

```
Actor iCleaner
  Maintain Goal MaintainBattery_loaded
    Maintain Cond chargingStatus &gt; 0.20
    Target Cond chargingStatus &gt; 0.98
  Maintain Goal Observe_Environment
    Maintain Cond empty ==false
  Maintain Goal PerformCleaning
    Maintain Cond empty ==true
  Achieve Goal TrashWaste
    Creation Cond full() ==true
    Creation Cond full() ==true
    Achievement Cond noneFull() ==true
  Achieve Goal ChargeBattery
    Context Cond isOn(actualposition) ==true
  Achieve Goal Locate_Wastebin
    Achievement Cond pos(type) ==actualpos()
    Drop Cond isEmpty() ==true
  Achieve Goal LocateCharging_Station
    Achievement Cond isOn(actualpos) ==true
    Drop Cond isEmpty() ==true
```

Figure 7.8: Goals with types and modelled conditions (right side, related entities are not displayed in this view) and the entities (artefacts) in the environment (left side).

Conditions on the environmental artifacts are first captured informally and then defined with the syntax of Java expressions, ready for the automated code generation. As an example, the goal TrashWaste (a sub-goal of DoCleaning in Figure 7.6) shall be adopted when the internal dirt box or the water bucket are full, and it will be achieved if the full one is empty again. In the following we report some of the modelled conditions.

Goal EnsureCleaning: failureCondition on Battery:Battery	chargingStatus < 0.01
Goal MaintainBattery_loaded: maintainCondition on Battery:Battery	chargingStatus > 0.20
Goal Observe_Environment: maintainCondition on KnownWastes:Waste	size() > 0
Goal PerformCleaning: maintainCondition on KnownWastes:Waste	size() == 0
Goal Locate_Wastebin: achieveCondition on WasteBins:WasteBin	isOn(my_location)==true
Goal TrashWaste: creationCondition on WaterBucket:DirtBox	full() ==true
Goal FindDirt: achieveCondition on KnownWastes:Waste	isEmpty() ==false

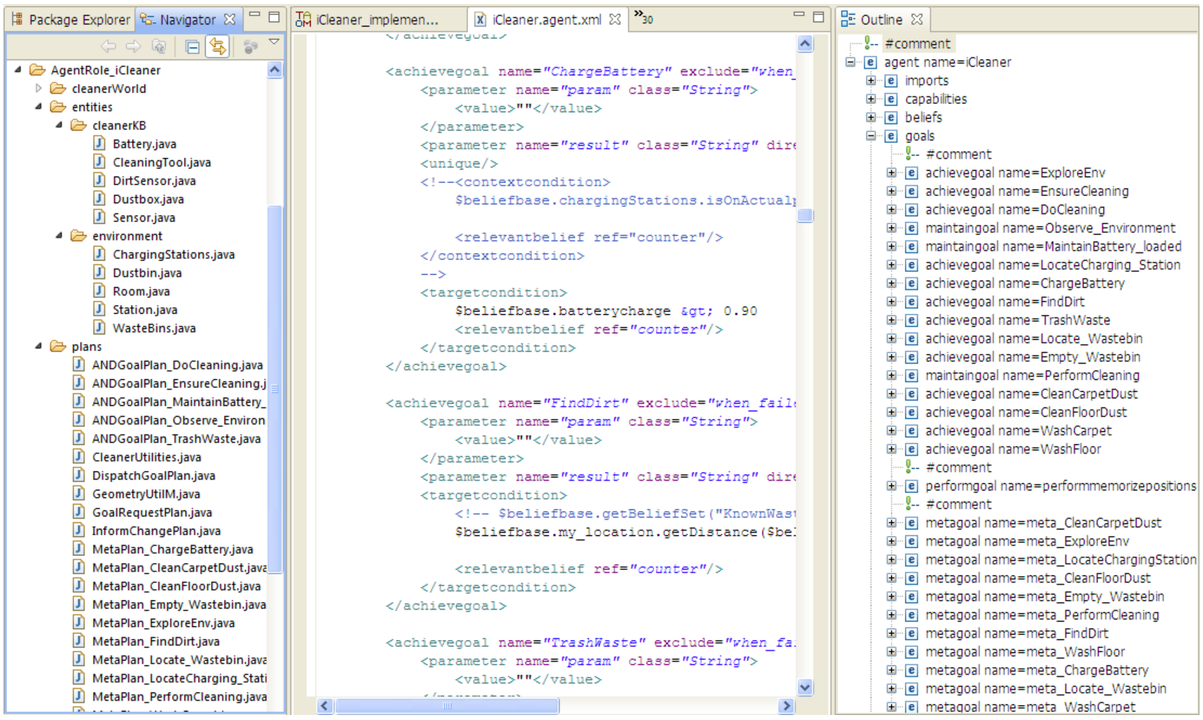
An excerpt of the conditions, modelled in the extended TAOM4E tool, can be seen in Figure 7.8, which displays also the defined goal types: three maintain-goals, the rest achieve-goals. Inhibition links are defined from MaintainBattery_loaded to DoCleaning and from ThrashWaste to PerformCleaning and Observe_Environment.

Mapping to the implementation The obtained models were mapped to a Jadex agent definition file (ADF) and Java code skeletons, by the ***t2x*** tool. The resulting ADF (>1200 lines of code, 16 goals and 36 plans, including auxiliary ones, for the last version of the goal model) contains the definition of the agent's belief base, available goals with their details, references to plans (which have to be implemented in Java), and accepted messages. The goal model structure, which guides the achievement of the high-level goals at run time and alternatives selection, is technically implemented by means of specialized auxiliary plans, together with a representation of the goal model in the agent's belief (Figure 7.9).

The example in Figure 7.10 shows a small part of the generated ADF for the *iCleaner* and the corresponding goal model part. The goal Observe_Environment is decomposed to two subgoals. This decomposition is annotated in the belief base (upper part of Figure 7.10) and handled by a dedicated plan. Plans (e.g. MoveToTarget) are handled by the Jadex goal triggering mechanism. Goal selection in means-end and OR-relationships is done by evaluation of softgoal contributions and the importance given to softgoals. For this system, we gave the same importance to each softgoal.

The interface to the simulation environment and the classes representing the ontological concepts were automatically generated from the above-mentioned ontology, by tools provided with Jadex.

Implementation and run-time After the automated code generation, the agent's plans (its *capabilities*, the concrete sensing and acting functionalities) were implemented



```
AgentRole_iCleaner
  cleanerWorld
  entities
    cleanerKB
      Battery.java
      CleaningTool.java
      DirtSensor.java
      Dustbox.java
      Sensor.java
    environment
      ChargingStations.java
      Dustbin.java
      Room.java
      Station.java
      WasteBins.java
  plans
    ANDGoalPlan_DoCleaning.java
    ANDGoalPlan_EnsureCleaning.j
    ANDGoalPlan_MaintainBattery_
    ANDGoalPlan_Observe_Environ
    ANDGoalPlan_TrashWaste.java
    CleanerUtilities.java
    DispatchGoalPlan.java
    GeometryUtilM.java
    GoalRequestPlan.java
    InformChangePlan.java
    MetaPlan_ChargeBattery.java
    MetaPlan_CleanCarpetDust.java
    MetaPlan_CleanFloorDust.java
    MetaPlan_Empty_Wastebin.java
    MetaPlan_ExploreEnv.java
    MetaPlan_FindDirt.java
    MetaPlan_Locate_Wastebin.java
    MetaPlan_LocateCharging_Stati
    MetaPlan_PerformCleaning.java
```

```
    <achievegoal name="ChargeBattery" exclude="when_
        <parameter name="param" class="String">
            <value>""</value>
        </parameter>
        <parameter name="result" class="String" dire
        <unique/>
        <!--<contextcondition>
            $beliefbase.chargingStations.isOnActual

            <relevantbelief ref="counter"/>
        </contextcondition>
        -->
        <targetcondition>
            $beliefbase.batterycharge &gt; 0.90
            <relevantbelief ref="counter"/>
        </targetcondition>
    </achievegoal>

    <achievegoal name="FindDirt" exclude="when_fail
        <parameter name="param" class="String">
            <value>""</value>
        </parameter>
        <parameter name="result" class="String" dire
        <targetcondition>
            <!-- $beliefbase.getBeliefSet("KnownWas
            $beliefbase.my_location.getDistance($be

            <relevantbelief ref="counter"/>
        </targetcondition>
    </achievegoal>

    <achievegoal name="TrashWaste" exclude="when_fa
        <parameter name="param" class="String">
            <value>""</value>
```

```
#comment
agent name=iCleaner
  imports
  capabilities
  beliefs
  goals
    #comment
    achievegoal name=ExploreEnv
    achievegoal name=EnsureCleaning
    achievegoal name=DoCleaning
    maintaingoal name=Observe_Environment
    maintaingoal name=MaintainBattery_loaded
    achievegoal name=LocateCharging_Station
    achievegoal name=ChargeBattery
    achievegoal name=FindDirt
    achievegoal name=TrashWaste
    achievegoal name=Locate_Wastebin
    achievegoal name=Empty_Wastebin
    maintaingoal name=PerformCleaning
    achievegoal name=CleanCarpetDust
    achievegoal name=CleanFloorDust
    achievegoal name=WashCarpet
    achievegoal name=WashFloor
    #comment
    performgoal name=performmemorizepositions
    #comment
    metagoal name=meta_CleanCarpetDust
    metagoal name=meta_ExploreEnv
    metagoal name=meta_LocateChargingStation
    metagoal name=meta_CleanFloorDust
    metagoal name=meta_Empty_Wastebin
    metagoal name=meta_PerformCleaning
    metagoal name=meta_FindDirt
    metagoal name=meta_WashFloor
    metagoal name=meta_ChargeBattery
    metagoal name=meta_Locate_Wastebin
    metagoal name=meta_WashCarpet
```

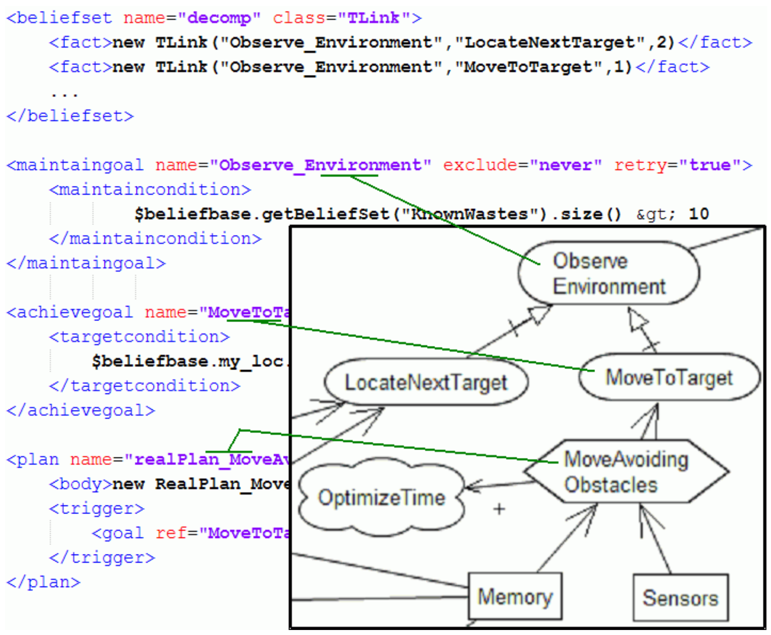
Figure 7.9: Excerpt of Jadex XML code and Java files (left side), generated with ***t2x***.

in the predisposed JAVA files. Moreover, the agent's belief base had to be adjusted for a correct access to the environment, and several settings in the ADF had to be performed.

One or more agents can be deployed on the simulation environment. They communicate with the environment by message passing and try to achieve their top goal, EnsureCleaning. Their knowledge of the environment can be displayed graphically (Figure 7.11). Furthermore, Jadex provides a visualisation of the goals dispatched, plans executed and messages sent at run-time, which can be useful for debugging.

Testing We adopted the agent testing tool *eCat* [Nguyen et al., 2008] for an automated testing of the iCleaner system for the achievement of its main goals, to obtain feedback to improveme not only of the implementation, but also the design and requirements models.

The tool consists of a test execution process with automated input generation with an evolutionary, ontology-based algorithm, evaluation and reporting functionalities. Testing proceeds without human intervention (and without time-consuming graphical interfaces), thus allowing to arbitrary extend testing time and to exercise and stress the agents under test as much as possible. A Tester agent generates new test cases (new



```
<beliefset name="decomp" class="TLink">
    <fact>new TLink("Observe_Environment","LocateNextTarget",2)</fact>
    <fact>new TLink("Observe_Environment","MoveToTarget",1)</fact>
    ...
</beliefset>

<maintaingoal name="Observe_Environment" exclude="never" retry="true">
    <maintaincondition>
            $beliefbase.getBeliefSet("KnownWastes").size() &gt; 10
    </maintaincondition>
</maintaingoal>

<achievegoal name="MoveToTa
    <targetcondition>
        $beliefbase.my_loc.
    </targetcondition>
</achievegoal>

<plan name="realPlan_MoveAv
    <body>new RealPlan_Move
    <trigger>
        <goal ref="MoveToTa
    </trigger>
</plan>
```

Figure 7.10: Part of a goal model and the corresponding generated Jadex ADF.

environments as well as dynamic changes in environments during a test) and executes the system on them. Monitoring agents monitor communication among agents and all events happening in the execution environments in order to trace and report errors. They are deployed transparently to the system under test, in order to avoid possible side effects.

In our case, for each version of the iCleaner, 1000 testcases of 30 seconds each were performed, measuring the performance in terms of waste removed, obstacles hit and battery failures. The generated environments differ for the number and placement of waste, obstacles, charging stations and waste bins.

7.2.3 Evolution of the iCleaner

We applied the *Tropos4AS* process along five iterations, resulting in five versions of the **iCleaner**. Testing results of the preceding version are used as a feedback which gives rise to new requirements and bug reports that were taken into account in the development of the subsequent version. Resembling a spiral model, the importance attributed to the different development phases varies in each iteration: in the first iterations, more importance is given to early phases, while later iterations focus more on detailed design and implementation. In the following we briefly describe the five documented iterations for the development of the **iCleaner**.

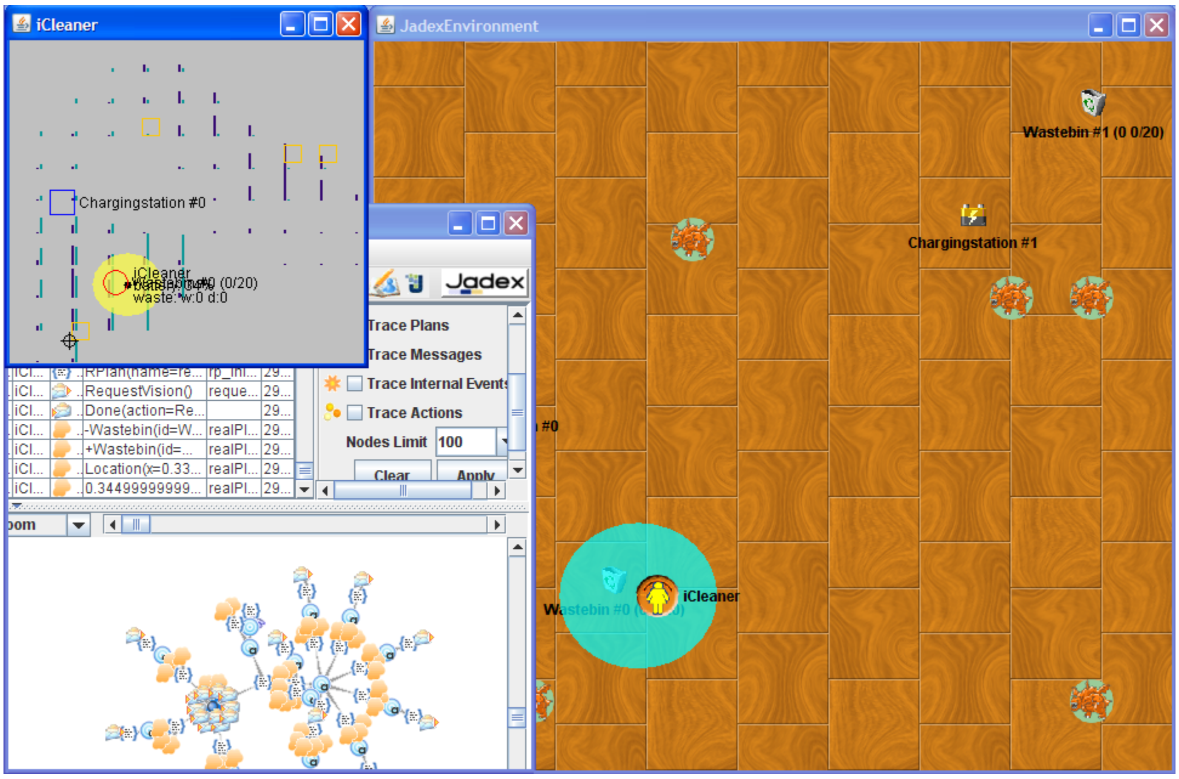


Figure 7.11: Run-time graphical simulation environment, the view of the iCleaner on it (upper left part) and the Jadex representation of the agent's internal goal achievement and plan execution behaviour (lower left part).

Version 1   The *iCleaner* goal model was extended, as illustrated in the previous section, with an environment model, conditions, goal types and relationships, before generating code with the ***t2x*** tool. Although this goal model was already discussed with the system designer, upon a detailed analysis of the expected run-time behaviour for goal achievement and on the corresponding agent's plans, different lacks were identified in the model, mainly regarding agent movement, whose specification was scattered in different plans and sensing activities in this initial model. Thus, we decided to limit this version to the evolution of the goal model and to enact changes to the model before spending specific programming effort on the generated code.

Version 2   A main change in the goal model for version 2 consisted in adding a dedicated goal and plan for movement MoveToTarget, whereas the plans deciding the target location remained scattered in different parts of the model. The decision on which target to select was made at a goal level, by defining inhibition links, e.g. between

MaintainBatteryLoaded and TrashWaste, to give precedence to battery loading.

The resulting goal model (which is similar in its structure to the one shown in Figure 7.12) leads to the following nominal run-time behaviour: the agent always tried to observe the environment, by locating new target destinations and moving to them. If there was some waste in the range of its cleaning tools, it was cleaned, either by absorbing or by mopping. After a revision of the conditions and a new code generation with ***t2x***, the plans were implemented and the goal and plan deliberation adapted to the needed behaviour, setting proper *Jadex* properties. This version was delivered to the testing step, with all behaviours implemented except obstacle avoidance and cleaning of wet dirt.

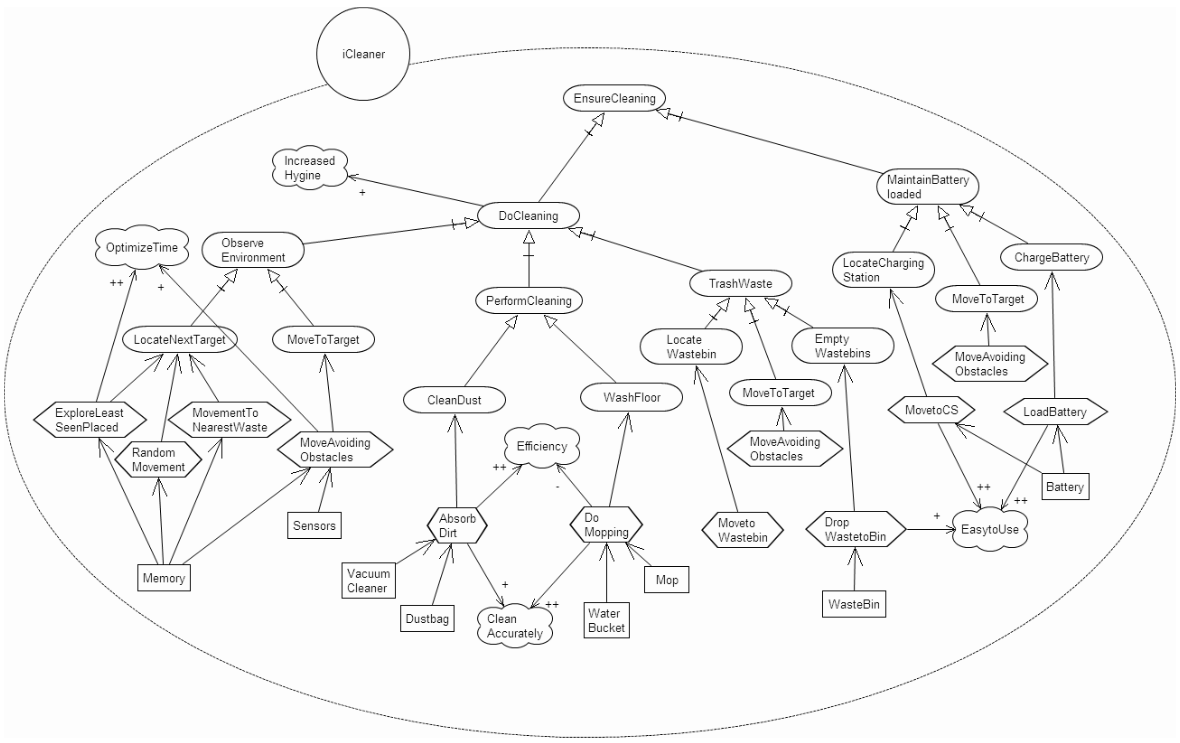


Figure 7.12: Goal model for the **iCleaner**, corresponding to the software implemented in version 5.

Version 3 For this version, the requirements did not change. By feedback from the testing, the code of different plans defining the agent movement towards battery loading stations and for emptying the cleaner's internal dust box into a waste bin were revised. Also, the code of plans for discovery of new charging stations and waste bins was improved by re-using the plan ExploreLeastSeenPlaces. Moreover, the previously missing avoidance of obstacles is implemented as requested in the requirements. This involved changes to the movement plan and memorization of the target destination while bypassing obstacles.

Version 4 To increase the efficiency of the **iCleaner** agent, plans for cleaning of wet waste were added to the goal model. Also, the wet waste has to be properly collected in the agent's internal water bucket and trashed to the waste bin. The updated goal model was re-mapped to the agent definition file and the plans were implemented, involving also changes to the goal achievement properties and conditions for the other two goals which are directly related to the cleaning activity.

Version 5 To further improve efficiency, a new alternative to achieve the goal LocateNextTarget was added: MoveToNearestWaste, which should be applied always if there exist waste perceived but not yet cleaned. For example, if some waste is perceived, but the internal dust box is full or the battery is too low, its position has to be added to the belief base. When the problem is resolved, the iCleaner moves towards the nearest known waste. The updates to the goal model (Figure 7.12) needed a new plan and belief set, which were implemented.

7.2.4 Testing results and improvements

Maintaining the battery loaded demands a change of the goals and of the behaviour of the agent (a simple form of adaptation), trying to find a free station where to recharge the battery if its level is low. Similarly, a full dustbag needs a change in the goal to achieve. Moreover, if the targeted waste bin is full, the agent has to properly adapt, searching a free one. Also, the detection of an obstacle in driving direction needs a temporary change of the target.

For testing the implemented agent for obtaining the desired behaviour, we use the following measures of efficiency and robustness, calculated over the test-cases run by the testing framework for each version: the dirt collected per test-case (efficiency), the number of crashes per test-case (robustness) and the total number of runs out of battery (robustness).

Figures 7.13 and 7.14 depict the improvement in efficiency and robustness of **iCleaner** throughout the versions. They report the average number of crashes with obstacles in the environment and the waste collected, respectively, computed on 1000 runs of the agent. We see that the number of crashes has reduced significantly in version 3, which is obvious since in version 2 the agent still has no capability to avoid obstacles. Further improvements in the next versions are the result of an optimization of distance parameters. However, for this reason the cleaning efficiency (Figure 7.14) decreases in version 3 because keeping a safe distance to objects and thus has to follow longer paths. The efficiency improved in the subsequent versions, adding the cleaning of wet waste and optimized movement strategies.

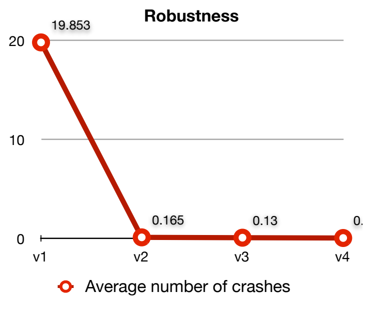


Figure 7.13: Crashes

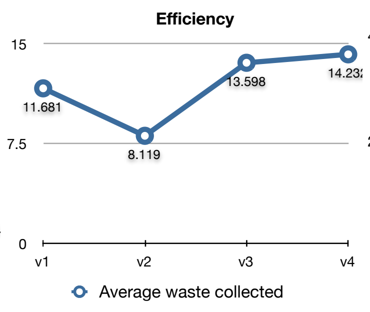


Figure 7.14: Waste

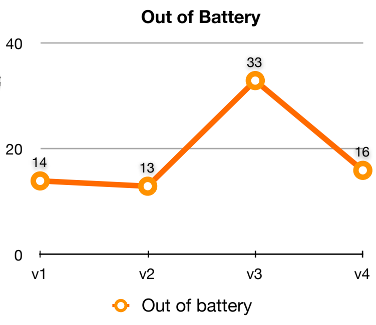


Figure 7.15: Battery

Figure 7.15 depicts the total number of times the agent runs out of battery in 1000 executions. In version 4 an unexpected value stands out: battery failures are nearly doubled. The value returns back to the previous level in version 5, although the behaviours regarding battery loading were not modified. We can explain this with the fact that the agent, cleaning more, has also to search more frequently for a waste bin. If the way from the *wastebin* to the battery loading station is far and obstructed by obstacles, the agent can easily run out of battery. In version 5, a new behaviour for moving to the nearest known waste is added. Therefore, the agent will prevalently move around in known area, reducing the probability of reaching hardly accessible places from which it is difficult to exit to reach the charging station in time.

7.2.5 Final Considerations

We developed a simple system following the *Tropos4AS* process until an implementation as goal-directed BDI agent. In five iterations of the process, the behaviour of the system was improved, basing on quantitative (by measurements for efficiency and robustness) and qualitative (by observation) feedback from an automated testing of the implemented prototypes. This feedback led to changes in the goal models, in the detailed design and the implementation.

Explicitly representing the goal model at run-time and guiding the agent behaviour directly with this goal model, wrong behaviours can be quickly localised, missing functionalities can be added seamless (as long as they have a well-delimited scope), and modifications can be enacted to the models and then mapped to the code.

The tool-support helps maintaining the consistency and traceability between goals and implemented functionalities and reduces the effort spent in each development phase and iteration. Similarly, the ontology is kept aligned with the code, as the classes for the domain entities are generated using tool support provided by Jadex. However, the tools miss round-trip engineering functionalities, which could improve the consistency and simplify changes on both sides. Also, the expertise requested on the tools, the modelling process and the implementation language, is quite high. Three subjects with different skills were involved in this case-study, for requirements engineering, detailed modelling and implementation, and testing.

The system exhibits basic adaptivity properties, which mainly arise from the interpretation of the extended goal model and the interplay of its goal types and conditions, at run-time. Thus it heavily differs from the example in Section 7.1, where adaptivity mainly arises from selection of alternatives and maximisation of softgoal contributions. The main focus was on implementing adaptivity features which directly arise form the knowledge captured in the goal models. More sophisticated forms of adaptivity would need an implementation of more complex reasoning and learning mechanisms, which have no direct correspondence in design time models and are thus out of the scope of this work.

We encountered various difficulties in the modelling and implementation of the system. Some of them led to improvements of the modelling process and of the code generation tool. Especially, the evaluation of conditions for guiding the goal deliberation behaviour of Jadex caused various problems, because the agent's beliefs were in some cases not updated in time. Also, the mapping of some goal types had to be adjusted to respect the intended meaning, by setting various flags in the goal definitions. Moreover, the interplay of the various goals which are active in parallel (e.g. maintain-goals) led to behaviours which were difficult to comprehend and thus challenging to debug and to maintain. This issue is common to various (e.g. agent-oriented, distributed) programming paradigms with more than one control flow and needs to be dealt with to promote these languages.

For future research it would be challenging to analyse how testing feedback can be interpreted from the system itself at run-time to adapt dynamically.

7.3 Evaluation: Feedback from Run-Time to the Design

In this sections we present two preliminary experiments, whose objective is twofold: on one hand, we aim at verifying the behaviour of the of a system created following the *Tropos4AS* process, with respect to the designed specifications; on the other hand, we give an outline for a **feedback mechanism from run-time to the design models**, to correct and to refine the system specification by exploiting information retrieved from an evaluation of the run-time behaviour of the system.

The envisioned process takes into account that new requirements may emerge also at run-time. A feedback of these requirements to the requirements analysis and design phases would be of crucial importance for the optimisation of a self-adaptive system. An effort in this direction can be seen as a first step towards a system which is able to interpret this feedback and to effect proper changes in its requirements specification and to adapt its behaviour at run-time.

The performed experiments start from a goal-oriented modelling and implementation following *Tropos4AS*. At run-time, changing user preferences guide the behaviour of the system simulation. An evaluation of the system execution leads to the refinement of the designed goal models (such as the modification or introduction of new relationships) upon the analysis of the system's run-time behaviour.

7.3.1 Outline of the applied feedback process

We develop a system following *Tropos4AS*, defining its requirements in a goal model. Variability, including desires and preferences of possibly changing users, is captured in goal models, e.g. by alternatives, softgoals and contributions to them. From the so-obtained models we automatically derive agent skeletons in the defined tool-based process. At run-time, the system (i.e. in this case, the obtained prototype) exhibits a certain behaviour in correspondence to variability in user desires and in environmental conditions. Users can communicate their needs and preferences to the system, e.g. via request messages. The resulting agent behaviours can be traced back to the specification of the alternatives in the goal model (since the agent is aware of and guided by its goal model at run-time), showing the effectiveness of the proposed framework in supporting traceability between run-time and design-time artefacts.

In a next step, the run-time behaviour of the system, in correspondence to the users' preferences, has to be analysed ad evaluated. In this step, the two small experiments, which are presented in the following, follow different paths. The first one relies on the opinion of experts for the definition of the model, while the analysis limits to a refinement of contribution values. In the second experiment, the users have to comment on the appropriateness of the obtained results. Moreover, in the second experiment the executed agent capabilities are also analysed quantitatively: capabilities used rarely can indicate a problem in the design of the decision making, or in the capability itself.

The results of this analysis can call for an improvement of the design artefacts. For instance, in the following preliminary experiments, contribution relationships between model elements will be corrected or further qualified. However, also goal refinements and operationalisations could be modified. Changes in goal models can be enacted at design time and re-mapped to code for a subsequent version of the software in an iterative process. In future, they could possibly also be effected on-line, directly to the run-time representation of the goal model (at the implementation level this is already supported for most of the artefacts in a goal model). We consider this work a first step towards setting up feedback mechanisms from run-time to design, a core aspect in developing self-adaptive systems.

7.3.2 Example 1: the Travel Agency

As first example to illustrate the idea of a feedback from run-time to the design artefacts, and also to show the effectiveness of reasoning on the goal model at run-time, we take a simple travel agency recommender system, TravelAgent, as modelled by a *Tropos* goal model in Figure 7.16. The system handles requests from several categories of customers (e.g. business customers, vacation customers or students) and gives proposals for a full travel package, according to the users' preferences. These preferences are modelled through softgoals. For example, as illustrated in Figure 7.16, possible softgoals to characterize a customer category are reasonable cost, good comfort, and relax vacation. The agent tries to achieve them exploiting different alternatives for journey and accommodation and selecting suitable additional activities.

The main idea is to focus on the preferences of different customer categories, which are recognized by the system at run-time by profiling users from the set of queries they submitted. We observe the system while it is adapting to each category, trying to maximize customer satisfaction (customer's softgoals delegated to the TravelAgent system) and providing evidence of how such softgoals have different impact to the system's own internal softgoals, e.g. maximize profit, which were delegated from the business stakeholders, e.g. from a travel agency. The models used in this evaluation are

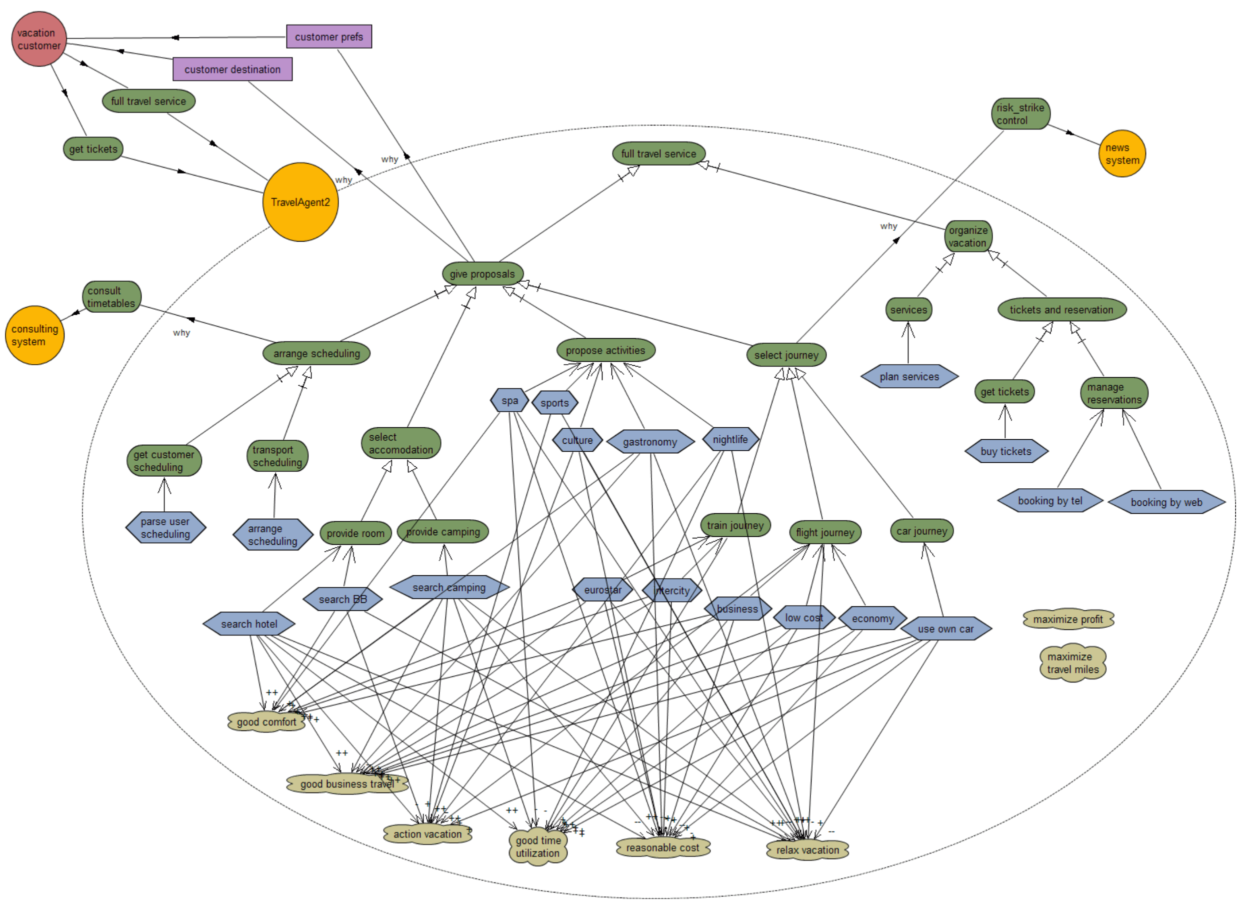


Figure 7.16: Goal model of the TravelAgent example with a detailed modelling of softgoal contributions.

Tropos goal models without the *Tropos4AS* extensions. Focussing on goal AND/OR hierarchies and softgoal contributions, we specifically evaluate the mapping to the implementation obtained by the *Tropos4AS* mapping to agent code using the ***t2x*** tool. In the following, we give a sketch of the experiment and the obtained results. Further details can be found in [Morandini et al., 2008e].

Experiment description

Suppose that a generic Customer can be distinguished into three categories: *business customer* (BC), *vacation customer* (VC) and *student customer* (SC), each one composed by individuals having similar preferences and similar requests to the travel agency system. For each category, an expert sets typical preference values, by varying softgoal importances (Table 7.1). A domain expert also defines contribution relationships between each capability and internal softgoals, such as maximize profit, see [Morandini et al., 2008e] for details.

$\mathbf{C_i}$	Vacation Customer	Student Customer	Business Customer
good_business_travel	0	0	1
good_comfort	0.6	0.2	1
action_vacation	0.4	1	0
good_time_utilization	0.3	0	1
reasonable_cost	0.6	1	0.1
relax_vacation	0.6	0.3	0

Table 7.1: Softgoal importances defined to profile user preferences in the three categories. These values are supposed to be given by domain experts.

In the simulation, the Customer (in the following also *user*) interacts with the system by submitting request messages that can be conceived as activation events for the system's goals. Moreover, the system is supposed to perform user profiling, analysing the queries, to recognize the category of a customer and and to adapt its selection criteria (i.e. the softgoal importances) accordingly. On the basis of this information the system is able to assume the best-suiting behaviour and thus to activate proper capabilities.

Table 7.2 shows the components of the process for the choice of capabilities. In particular, the 2nd column represents the input elements. The possible queries related to a customer class are denoted as, e.g. q^{BC} for a business customer query, whereas C^{BC} holds the set of user preferences and constraints denoting a user category, perceived by the system via user profiling activities or by user-guided configuration. The system chooses an appropriate set of appropriate behaviours, i.e. a set of leaf-level goals, for the query and constraints in input, denoted in the 3rd column, e.g. b_i^{BC}. For a behaviour b_i, Cp_i (column 4) denotes the possible sets of capabilities (i.e., here, sets of plans) an agent can execute to exhibit this behaviour. The system is able to compute the set of possible behaviours b_{1-m}^{BC} that it can exploit in order to accomplish user requests, maximizing their preferences, and to retrieve the capabilities that have to be activated in order to operatively execute the chosen behaviour.

We prepared the experiments, defining a set of queries for every different customer category. Table 7.3 gives examples for sets of queries for the three classes of users we considered.

User class	Trigger events	Behaviours	Capabilitiy sets
BC	q_1^{BC}, C^{BC}	b_1^{BC}	Cp_1^{BC}
	...	...	...
	q_m^{BC}, C^{BC}	b_m^{BC}	Cp_m^{BC}
VC	q_1^{VC}, C^{VC}	b_1^{VC}	Cp_1^{VC}
	...	...	...
	q_n^{VC}, C^{VC}	b_n^{VC}	Cp_n^{VC}
SC	q_1^{SC}, C^{SC}	b_1^{SC}	Cp_1^{SC}
	...	...	...
	q_k^{SC}, C^{SC}	b_k^{SC}	Cp_k^{SC}

Table 7.2: System inputs (Events and constraints) and outputs (behaviours and sets of capabilities) for the simulation

Query	Vacation Customer	Student Customer	Business Customer
q_1	give proposals	give proposals	give proposals
q_2	provide camping, car journey, prop. act.	propose activities	provide room, flight journey
q_3	provide room, train journey	provide room, flight journey	select journey
q_4	select accommodation	camping, train journey	select accommodation

Table 7.3: Queries that characterize each customer category.

Analysis of results

After the simulation, the data related to the experiment has been collected. According to our first objective, we are able to monitor the system behaviour (b), each time a query (e.g. q_4^{BC}) occurs, along with some user preferences (e.g. C^{BC}={good comfort}), verifying that b coreectly belongs to the goal model. Specifically, we observe that the system has the ability to adapt its behaviour to accommodate with the current customer category. For example, assuming that q_4^{BC} will trigger the goal select accommodation, along with giving preference to good comfort. Now, the system is able to navigate the goal model in order to maximize the softgoal which captures this user preference.

Looking at the goal model illustrated in Figure 7.16, we can see that the goal select accommodation has two alternative ways to be achieved, i.e. provide room and provide camping. The system will first try to select provide room, because its capabilities (characterized by the two plans search hotel and search BB) give the biggest contribution to the given user preference. The same procedure will be used in a next step to discriminate between the two available capabilities, this time resulting in the selection of search hotel. These experiments confirm the ability of the framework in supporting traceability between run-time and design-time artefacts.

To meet our second objective, we simulate the execution of a set of user queries and preferences in order to revise softgoal relationships in the goal model. Table 7.4 shows the sets of capabilities activated by the system, i.e. the behaviour instances it selected at run-time, as a response to the simulated user queries described in Table 7.3. In Table 7.4, each row specifies a query from a particular category of users (BC, VC and SC). Contributions to *maximize profit* are calculated by summing the values of contribution (defined by the domain experts) to the executed capabilities, e.g. for Cp_1^{BC}: *eurostar train* + *search hotel* + *gastronomy* $= 0.2 + 0.8 + 0 = 1$. Notice that an analysis by simulation does not cope with the possible contribution produced by all the different capability groupings. On the contrary, the simulation will converge towards the only sets of capabilities requested by the real customer categories.

Query	Sets of executed capabilities	Contribution to *maximize profit*	
q_1^{BC}	Cp_1^{BC}: eurostar train, search hotel, gastronomy	1	
q_2^{BC}	Cp_2^{BC}: business flight, search hotel	1.7	
q_3^{BC}	Cp_3^{BC}: eurostar train	0.2	
q_4^{BC}	Cp_4^{BC}: search hotel	0.8	$avg = 0.925$
q_1^{VC}	Cp_1^{VC}: low cost flight, search BB, culture	0.6	
q_2^{VC}	Cp_2^{VC}: use own car, camping	0.3	
q_3^{VC}	Cp_3^{VC}: intercity train, search BB	0.5	
q_4^{VC}	Cp_4^{VC}: search BB	0.4	$avg = 0.45$
q_1^{SC}	Cp_1^{SC}: low cost flight, search camping, nightlife	0.5	
q_2^{SC}	Cp_2^{SC}: nightlife	0	
q_3^{SC}	Cp_3^{SC}: low cost flight, search BB	0.6	
q_4^{SC}	Cp_4^{SC}: intercity train, camping	0.4	$avg = 0.375$

Table 7.4: Capability groups associated to every query at run-time.

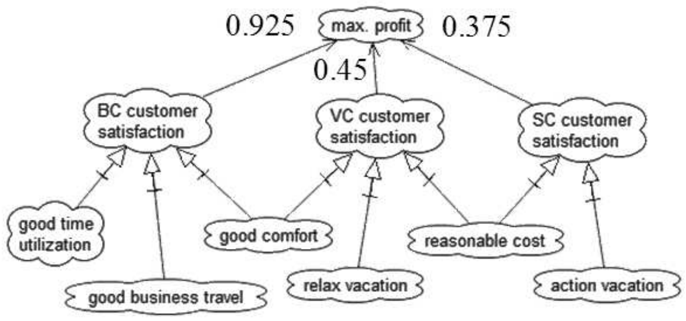
With the results of these queries we can observe how (in our case simulated) customer preferences affect system behaviour. The capability groups corresponding to the different behaviours of the TravelAgent can then be used to add or quantify *Tropos* contribution links.

Figure 7.17 A), shows the averages f_{avg} of the values obtained in Table 7.4, considering the internal softgoal maximize profit. In Figure 7.17 B), a softgoal customer satisfaction is introduced to aggregate the softgoals relevant to a specific customer category. Contributions between them and the internal softgoal maximize profit can be drawn and quantified by the contribution values computed at run-time. This result can contribute both to validate existing contribution links and to add new ones. In the case run-time feedback is in contrast with the design-time models, a revision of the goal model could be required.

In a subsequent step, these new relations could be used by the system to adapt its strategic behaviours, not only according to the user preferences (i.e. softgoal customer

Customer Satisfaction	Contribution to *maximize profit*
BC	0.925
VC	0.45
SC	0.375

A)



B)

Figure 7.17: A) Resulting quantitative contributions between customer categories and the softgoal maximize profit; B) visualizing the results in terms of a goal model refinement. The labels define the new contribution values.

satisfaction), but also according to its internal organizational objectives (i.e. softgoal maximize profit), following a trade-off for the achievement of these two softgoals.

7.3.3 Example 2: a computer recommender system

As second example for an implementation following the *Tropos4AS* automanted mapping from goal models to code and for evaluating the idea of feedback from run-time to the design artefacts (or for a modification of the implemented goal models at run-time), we present a *computer recommender agent.*

The computer recommender agent handles requests from customers and gives proposals for several computer compositions, according to the users' input requests. As in the previous example, in the *Tropos4AS* goal model these preferences are modelled through softgoals (see Figure 7.18 for a reduced example). Softgoals that characterize an user's preferences are, for example, high performance, storage space, simplicity of usage, versatility and compactness. The agent tries to recommend a computer system, selecting from the various configurations available in commerce[1], exploiting different alternatives for processor, memory, graphic card and additional services.

In the following, we give a sketch of the experiment and the obtained results. Further details on the implementation, the survey and its evaluation can be found in [Tomasi, 2009].

[1]The prototype accesses to a locally stored version of the DELL website to search for suitable configurations

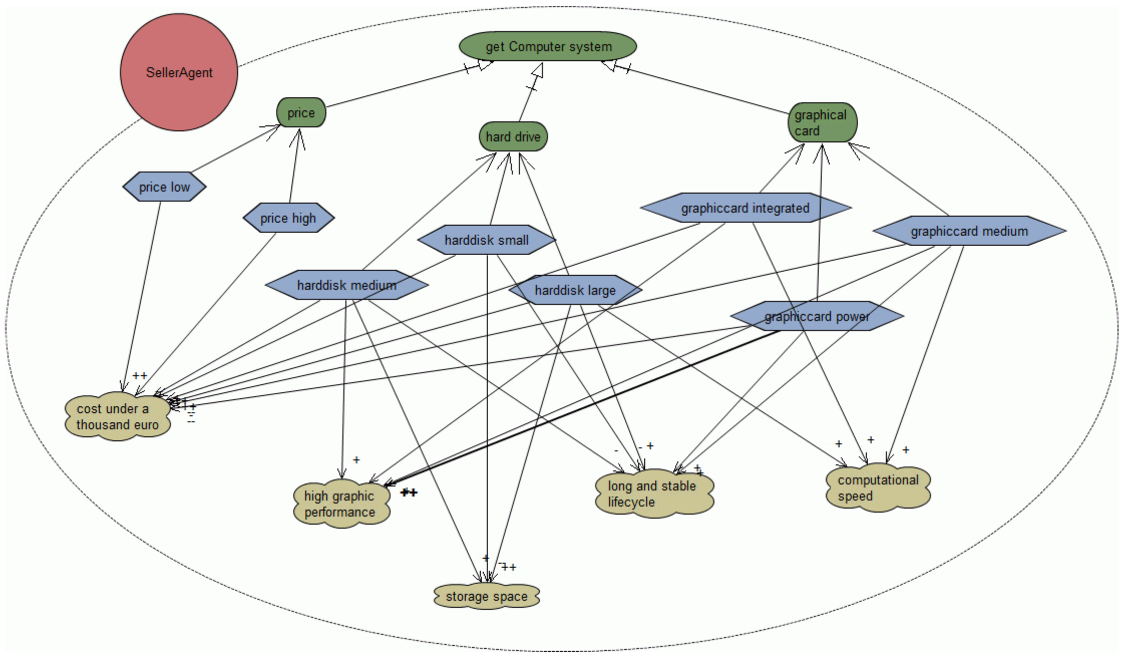


Figure 7.18: Part of the Computer Seller System goal model.

Experiment description

The system is implemented in Jadex, basing on the automated mapping, with the goal model represented at run-time. At run-time, the modelled softgoals (with their positive or negative contribution relationships) build the main link between the system and its users. Users who need a recommendation for a new computer, make a request to the system, expressing their preferences by filling a questionnaire. This questionnaire is directly related to the modelled softgoals, thus the user gives different weigh to softgoals, defining selection criteria for available alternatives.

To achieve the user's request for a computer system, the system tries to find an operationalisation (i.e. a set of capabilities Cp) which satisfies this goal and maximizes the contribution to the softgoals which are important for the user. Basing on the selected capabilities, which determine computer configurations, the system then selects the best-suited systems from its database, which it proposes to the user. In fact, the system performs only a single traversal of the goal model per user request, but this simple mechanism suffices to motivate the following feedback process.

The proposals are then given to the users, which rate them with one out of the three keywords "satisfied", "oversized" and "undersized". Figure 7.19 gives an example of preference values given in input to the system and feedback values obtained from the users for the selected system. This evaluation was performed with 20 participants from

Softgoals	**User1**	**User2**	**User3**	**User4**	**...**
price under a thousand euro	0,8	1	1	0.8	
high graphic performances	0,9	0.5	0.1	0.6	
computational speed	0,8	0.8	0.3	0.8	
run clear and fast	0.7	1	1	0.5	
simple to use	0,6	0.8	1	0.5	
storage space	1	0.5	1	1	

FEEDBACK	**S**atisfied	**S**atisfied	**O**versized	**U**ndersized	**...**

Figure 7.19: Preference values in input and feedback values for the recommended computer systems, for different users.

various user categories (although many of them were experienced computer users).

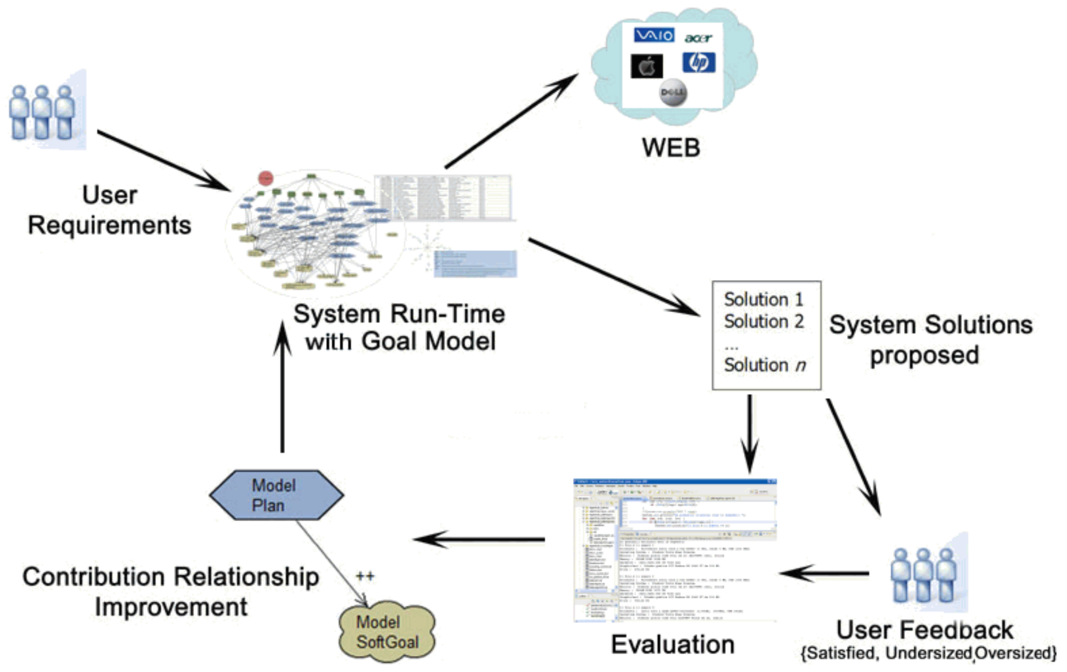

WEB
User Requirements
System Run-Time with Goal Model
Solution 1
Solution 2
...
Solution *n*
System Solutions proposed
Model Plan
++
Model SoftGoal
Contribution Relationship Improvement
Evaluation
User Feedback
{Satisfied, Undersized,Oversized}

Figure 7.20: Illustration of the user-driven feedback process.

As next step in the feedback process sketched in Figure 7.20, the obtained data has to be evaluated. Recommendations which were not satisfactory for the users, have to be examined in detail, analysing the preferences given and the capabilities (i.e. plans) activated, and trying to find the problematic points in the model, typically improper softgoal contributions. A second source of data for a possible improvement of the models is a global analysis of plan activations. Plans which were rarely or never activated can indicate problems in the decision algorithms (e.g. wrong softgoal contributions) or capabilities which are not useful for the system (and which thus could be deprecated in further versions of the software). However, in different domains such capabilities could also denote critical exceptional situations. In this case they should be properly captured in failure models, as presented in Section 3.3.2 of this thesis.

Necessary changes in a system can be carried out in the design time goal model, or else directly in the goal model represented at run-time, e.g. for softgoal contributions, by performing changes to the agent's belief base. By the evaluation we recognized some needs for contribution change. They were effected and the problematic user queries were repeated. While part of the users now gave a satisfactory feedback, others did not. However, these users were also not satisfiable with a manually selected computer, since we limited the system to offers from a single manufacturer. Moreover, the evaluation was performed manually and required a deep knowledge of the system and the domain. Repeating the tests, introducing, for testing purpose, various artificial errors, we could also observe that such models, which are considerably complex, are quite resilient to errors, and thus it remains difficult to localize errors. Our initial aim, a statistical analysis for the discovery of correlations between unsatisfied users and inadequate contribution links, would require a much higher number of subjects.

7.3.4 Contributions

In this section we presented two case studies with the aim to show the behaviour of a system aware of its goal model at run-time, and, specifically, to give a first sketch for feedback mechanisms from run-time to the design time artefacts (i.e. the goal model in our case) to correct errors in the requirements and to accommodate requirements changes emerged at run-time.

With a simple travel agency example, implemented with the help of the ***t2x*** tool, we gave a first approach on how to refine a goal model by information acquired from the execution of the software, which navigates its goal model at run-time, adapting to users' needs and preferences. The approach was tested by simulating an environment where several categories of users, which are characterized by their own preferences, interact with the system, requesting a service. Starting from the run-time behaviour of the system in response to user queries, a way to refine the relationships among a set of user preferences and the preferences (softgoals) of the business stakeholders, was described. This information can then be considered by a revised version of the system, to achieve a behaviour that better satisfies the stakeholders' preferences. This study preceded and inspired the introduction of some of the *Tropos4AS* modelling extensions, emphasizing the need for obtaining systems with more dynamic forms of adaptivity.

The second case study, performed in the context of a master thesis [Tomasi, 2009], follows a slightly different direction: users of a system are asked for their satisfaction with the given service (in this case a suggestion for a suitable computer system). This feedback is used, together with the user's preferences and usage statistics, for an analysis of problems in the requirements model. This study is limited with respect to the analysis of the feedback and should moreover be extended to systems with a more complex goal satisfaction behaviour, integrating also *Tropos4AS* modelling extensions. However, it is worth mentioning, because it points out critical issues such as the the resilience of the system to small errors and the number of participants necessary for an evaluation based on statistical results.

With these case studies a first investigation was carried out, for bringing information gathered at run-time back to the requirements and design phases, to contribute to the general problem of requirements and software evolution.

Chapter 8

Empirical Evaluation of Tropos4AS Modelling

8.1 Introduction

The *Tropos4AS* framework, presented in Chapter 3, introduces various extensions to the agent-oriented methodology *Tropos*, aimed to support the development of self-adaptive systems. In this section we present an evaluation of the *Tropos4AS* modelling language in comparison to the underlying *Tropos* modelling language[1], performed by running an empirical study, consisting of modelling and comprehension tasks performed by a group of researchers and students.

The evaluation of a modelling language can be characterised by three main aspects: (1) the effort for modelling, (2) the effectiveness of modelling (i.e. the expressiveness of the obtained models) and (3) the comprehensibility of the models. The empirical study is subdivided into two experiments:

- First, we evaluate if *Tropos4AS* is effective in modelling self-adaptive systems, with an acceptable modelling effort, in comparison to *Tropos*.
- Second, we evaluate if the *Tropos4AS* modelling extensions increase the comprehensibility of the requirements of a system.

Along these experiments, we are also interested in understanding how software engineers feel with and use the *Tropos4AS* modelling extensions. The design of the experiments follows the guidelines by Wohlin et al. [Wohlin et al., 2000] on how to set

[1]We refer to the *Tropos* modelling language, as defined in [Bresciani et al., 2004a, Susi et al., 2005]. In particular, we focus on *Tropos goal diagrams*, which are mainly affected by the novel extensions.

up and document empirical studies in software engineering. It allows to have a high degree of control over the study, to achieve results with statistical significance.

Discussion on the study set-up A comparison of *Tropos4AS* with a similar engineering methodology (except *Tropos*), from a methodological point of view, would inevitably also assess the performance of the whole *Tropos* language. Thus, an evaluation limited to the novel extensions proposed in this thesis, which is our scope, would be impossible.

Tropos4AS covers the whole development cycle until the implementation. Thus, a comparison with code written without methodological aid (or with code developed with a cut-down version of the *Tropos4AS* methodology, which limits to concepts that are available one-to-one in the implementation language *Jadex*) would give significant results for the efficiency of the framework. However, such an empirical study, which involves implementation, would require participants that are experienced in the use of the implementation language (which is, in our case, Jadex) and, in particular, would demand an unacceptably high time effort for them.

Moreover, the experimental setup is guided to some extent by the availability and the experience of the potential *subjects* of the study (i.e. the participants). This influences the choice of the treatments that will be used for the comparison (in this study, the methodologies to compare), of the *objects* under study (i.e. the systems that are modelled) and the tasks to perform in the experiment.

Considering these constraints, we decided to compare *Tropos4AS* with its underlying methodology *Tropos*, focussing on the modelling aspects. A comparison of the whole *modelling process* between the two treatments *Tropos4AS* and *Tropos* would not have been feasible within the given time constraints, on a non-trivial model.

We are specifically interested in studying the use that the subjects make of the available modelling concepts for representing requirements, and in evaluating the effort and efficiency of modelling. Thus, our **first** aim is to compare the two *modelling languages*, by performing an off-line experiment that demands from the subjects the construction and analysis of the respective models. To ensure a fair comparison despite the higher expressiveness of the *Tropos4AS* language, we address experimental tasks which can be performed with both languages, satisfactorily.

A drawback in an experiment where the participants have to model some part of a system is, that the obtained models are typically difficult to be compared, since requirements modelling is a creative process which has not a single correct solution[2]. The

[2]Note that it is not our aim to have a complete, formal representation of requirements, for whose

analysis has therefore to be mainly based on subjective results obtained by questionnaires and by a high-level analysis of the obtained diagrams.

Second, we examine the comprehensibility of models created by applying the two methodologies. In this study of model comprehensibility the subjects have to answer to various questions, analysing models prepared by the researchers, understanding them, and extracting information. Such a study which limits to model comprehension ensures a higher controllability of the experiment execution and is thus able to provide more objective measurements, e.g. by calculating precision and recall (see page 189) of the answers given, with respects to the set of expected answers.

8.2 Experiment planning

8.2.1 Goal of the study

The goal of this empirical study is to compare the *Tropos4AS* modelling language, presented in Section 3.2 in this thesis, to the *Tropos* modelling language, to show if the benefits expected from *Tropos4AS* modelling are present also if used by both non-experts and experts of *Tropos*, in a realistic environment. Hence, the *main factor* of both experiments will be the modelling approach used: *Tropos* and *Tropos4AS*, the *treatments* that we want to compare.

8.2.2 Context selection

The experiment is run in a research centre with participants (researchers, doctorate students and programmers) from the software engineering group[3]. The experiment is run off-line, as a blocked subject-object study [Wohlin et al., 2000, Chapter 5]. The two objects, requirements specifications of two software systems, have been assigned to each of the participants (the subjects).

8.2.3 Objects of Study

The experiment has to be fair, not giving disadvantage to one of the two methodologies, to achieve expressive results. In our case, one methodology completely includes the other. The difficulty consists in selecting modelling tasks that can be performed with both methodologies and are not only tailored towards the new extensions, but that are

correctness can be verified, as e.g. in *Formal Tropos* [Fuxman et al., 2001].

[3]`http://se.fbk.eu`

challenging enough to prompt for the use of the novel extensions, whenever they are available.

In our work we define *self-adaptivity* as the ability to automatically take the correct actions, based on their knowledge of what is happening in the operating environment, guided by *objectives* assigned by the stakeholders (cf. our definition in Section 1). *Self-adaptivity* can not be easily synthesised to a single property of a sample system. Taking this into account, we provide two small system specifications as objects of this study. Covering as much as possible the definition, they provide a normal as well as some exceptional behaviour, and some circumstances drive the system to change its behaviour, to satisfy its requirements.

The objects of the study are two small, imaginary software systems, a Patient Monitoring Agent (PMA) and a Washing Machine Manager (WMM). PMA is a system that monitors elderly people in a smart home, for taking meals and medicine. WMM explains an intelligent washing machine controller, which adapts the washing settings to the user's preferences for energy saving and cleanness. The detailed requirements for each system are defined in system stories, reported in Appendix A.

8.2.4 Subjects

The subjects of the experiment are 12 employees of the research centre (which were not involved in the preparation and pilot run of the study), 6 researchers and 6 doctorate students and programmers. Only part of them are known to have used goal-oriented modelling languages (mainly *Tropos*), before.

8.2.5 Experiment design

We adopt a **paired, counterbalanced** experiment with two laboratories. In this experiment design, each subject has to perform the experimental task with both objects, and with both treatments. Moreover, half of them have to use e.g. the PMA system for the first experimental task (1st laboratory) and the WMM system for the second; vice-versa for the others. This design mitigates learning effects between the two treatments and between the two objects.

The subjects are randomly divided into 4 groups of 3 people each, to whose the treatments and the two objects, PMA and WMM, are associated as defined in Table 8.1. The paired design, having the same number of subjects in each of these groups, enables a better comparison and the application of more precise statistical methods.

	1st laboratory	2nd laboratory
Group 1A	PMA with *Tropos*	WMM with *Tropos4AS*
Group 1B	WMM with *Tropos4AS*	PMA with *Tropos*
Group 2A	PMA with *Tropos4AS*	WMM with *Tropos*
Group 2B	WMM with *Tropos*	PMA with *Tropos4AS*

Table 8.1: Assignment of subject groups to laboratories, objects, and treatments.

8.3 Experiment 1: Modelling

In the following, we show the design, procedure, analysis and results of the modelling experiment performed.

8.3.1 Research questions and hypotheses

With the goal of evaluating the benefits of *Tropos4AS* for modelling self-adaptive systems, in comparison to traditional *Tropos* modelling, we define two research questions.

*RQ*1**:** Is the effort of modelling requirements with *Tropos4AS* significantly higher than the effort of modelling them with *Tropos*?

*RQ*2**:** Is the effectiveness of *Tropos4AS* models significantly higher than the effectiveness of *Tropos* models, for representing requirements of an adaptive system?

As a basis for statistical analysis of the experiment, each research question has been translated to the corresponding *null-hypothesis* H_0 and *alternative hypothesis* H_a. The *null-hypothesis* typically denotes that the treatment has no significant effect on the result. Thus, the experimenter typically wants to reject it with high statistical significance. The null-hypothesis will be rejected in favour of the *alternative hypothesis*, which denotes that there exists a significant effect of the different treatment.

We frankly expect that the extended language needs a higher modelling time than the base language. However, we want to understand if the additional effort and the model complexity is still perceived to be reasonable. On the other side, we want to verify that *Tropos4AS* is more effective in modelling, obtaining models that reflect the requirements in a more complete way, and that would better support the developers in the implementation. These two assumptions lead us to the following hypotheses and define their direction (i.e., the hypotheses are one-tailed). For *RQ*1 we define:

- $H_0 1$: The effort of modelling requirements with *Tropos4AS* is not significantly higher than the effort of modelling with *Tropos*.

- H_a1: The effort of modelling requirements with *Tropos4AS* is significantly higher than the effort of modelling with *Tropos*.

whereas the following are the null-hypothesis and alternative hypothesis relative to *RQ*2:

- H_02: The effectiveness of *Tropos4AS* models is not significantly higher than the effectiveness of *Tropos* models.
- H_a2: The effectiveness of *Tropos4AS* models is significantly higher than the effectiveness of *Tropos* models.

Aspects characterising the research questions The two research questions include the abstract terms *effectiveness* and *effort*, which have to be detailed in order to associate them to variables that can be evaluated in the experiment. We decompose *RQ*1 and *RQ*2 to various aspects that characterise them and therefore define the terms *effectiveness* and the *effort* for the scope of the study. *RQ*1 is decomposed to aspects considering the time spent, the effort perceived by the subjects, and the difficulties encountered by the subjects during modelling. In detail they are:

(a1) overall time consumed for the modelling task

(a2) adequateness of the effort required for creating the models: We want to know if the modelling effort is subjectively perceived to be adequate by the subjects.

   a) required overall effort for modelling
   b) adequateness of the additional modelling effort specifically needed by Tropos4AS modelling in comparison with *Tropos*.

(a3) effort distribution: We want to study the changes in the distribution of the time spent on the following activities.

   a) reading modelling language specification
   b) understanding the example
   c) modelling the example

(a4) difficulties encountered in modelling: We want to know if the subjects perceived any difficulties in modelling the examples and in using the concepts of the modelling language.

   a) difficulty of modelling all the example details
   b) difficulty of using the modelling language

The aspects for $RQ2$ consider the expressiveness of the modelling language, perceived by the subjects, and the correctness of the models built:

(a5) perceived expressiveness of the modelling language: We are interested in the subject's judgement on the adequateness on the concepts provided by the language, for describing the requirements.

   a) effectivity in capturing the requirements
   b) adequateness of the modelling concepts

(a6) perceived effectiveness of modelling for an implementation: The subjects, as potential users, should comment on the utility of the models for software architects and programmers.

(a7) perceived utility of extensions for modelling adaptivity: We would like to know the opinion of the subjects on the single extensions introduced with *Tropos4AS*.

   a) modelling conditions
   b) modelling failures
   c) overall

(a8) measured correctness of the models drawn by the subjects, tested by evaluating the coverage of various scenarios, to know if the modelling concepts were used in a correct way and to which extent the models cover the requiements.

Each aspect has been investigated with its own research question (and associated hypotheses), of the same form and direction as the high-level questions $RQ1$ and $RQ2$. For instance, let us consider the aspect (a1). Its research question is $RQa1$: "Is the time required to model requirements with *Tropos4AS* higher than the time required with *Tropos*?".

8.3.2 Variables and measures

The *independent variable* and *main factor* of the study is the modelling language used to model requirements, considered with the two treatments *Tropos* and *Tropos4AS*. This variable is manipulated and controlled and should be independent from the objects, subjects, and experiment tasks, by the design of the experiment (see Section 8.5.5).

The *dependent variables* are the 8 aspect (a1,..., a8) identified and evaluated my means of questionnaires filled by the subjects before and after the experimental task. The continuous variables a1 and a3, are associated to the questionnaire, measuring the time spent for the whole experiment and fractions of time (in %) spent for various activities, and the *Likert scale* variables a2 and a4,..., a7.

Likert scale variables specify the level of agreement to a statement. They are defined, in our case, on an ordinal scale from 1 to 5, as follows: 1 strongly agree; 2 agree; 3 not certain (neutral answer); 4 disagree; 5: strongly disagree. The discrete variable a8 is evaluated on a scale from 0 to 4 by an expert, evaluating the correctness[4] of the models by counting the relevant aspects identified for covering two predefined execution scenarios.

8.3.3 Experiment procedure and material

Experiment procedure The experiment consists of the following steps:

1. Tutorial on *Tropos* and *Tropos4AS*
2. Pre-questionnaire
3. Laboratory 1
4. Questionnaire for laboratory 1
5. Laboratory 2
6. Questionnaire for laboratory 2
7. Post-questionnaire

Since the subjects had different levels of experience with *Tropos* and *Tropos4AS*, to prepare them for the experiment we gave a tutorial of 90 minutes on *Tropos* and *Tropos4AS* modelling, some days before the experiment. The tutorial presentation slides were sent to all participants, to be used also during the experiment, if needed.

The questionnaires and laboratories is done individually by each participant, at their desk, in a time frame of approximately three hours. Only after completing the first laboratory, the documents for the second laboratory are handed over.

[4]Note that we are not aiming for a verification of the formal correctness of the models in respect to the requirements specifications.

Prepared input material To perform the experiment, each participant receives the following documents (a complete sample is shown in Appendix A): a detailed description of the experiment procedure, the pre-questionnaire, the post-questionnaire, and, for each of the two laboratories, the following material:

- A summary of the modelling language to use (either *Tropos* or *Tropos4AS*).
- The requirements specification of the system to model (WMM or PMA).
- A sheet with an outline of the goal model, to be used to draw the model, and two control questions, useful for the subjects to cross-check the models.
- The relative questionnaire.

Experiment task The subjects have to model the two systems, one with *Tropos*, and one with *Tropos4AS*, as assigned to them (Table 8.1). Each system should be modelled with as much details as possible, with the methodology assigned to it, following step by step the scenario description. To eliminate the training effort with the TAOM4E modelling tool [Morandini et al., 2008b], with its constraints and usability issues, and thus to eliminate the influence of a possible further threat, we opted for drawing the diagrams on paper.

Each system description includes two *control questions* (an example can be found on page 228, Appendix A), which should be answered by the subjects, specifying the plans executed in a certain situation. These questions are not evaluated, but should be used by the subjects to cross-check the models. Before the tasks, after each laboratory and at the end, the corresponding questionnaires should be filled. A collection of the questions in the three questionnaires is listed in Table 8.2.

The pre-questionnaire asks for name and position of the participants, and a signature for authorisation for the collection of sensitive data. Moreover, some questions are made ([preq12]...[preq17] at the top of Table 8.2), which help to evaluate the knowledge of the two methodologies, and thus the adequateness of the tutorial.

The questionnaire associated to each of the laboratories includes the questions [q4]...[q17] in Table 8.2, which analyse the adequateness of the objects and the time, and collect the subject's perceptions for the specific treatment applied. Moreover, the overall time needed for completing the experimental task of each laboratory, should be recorded (except questionnaire filling). The participants are also asked to keep track of the time fractions (in %) spent for the various activities, and specifically for modelling the *Tropos4AS* extensions, as reported in Table reftable:questtime, questions [*t*], [q1]...[q3], and [q13]...[q15]. An indicative time of 1h is given as a suggestion for performing each laboratory, but the participants are free to take the time they need.

□1 strongly agree □2 agree □3 not certain □4 disagree □5 strongly disagr.		
preq12	Experience with requirements analysis	□Few □Research □Industry
preq13	Experience with *Tropos* modelling	□None □Small □Experienced
preq15	I understood the basic notions of *Tropos* modelling	□1 □2 □3 □4 □5
preq16	The visaul notation used in *Tropos* is clear	□1 □2 □3 □4 □5
preq17	The visual notation used in *Tropos4AS* is clear	□1 □2 □3 □4 □5
q4	The explanation of the example was clear to me	□1 □2 □3 □4 □5
q5	I had no difficulties in modelling its requirements in a goal model	...
q6	I had enough time for accomplishing the modelling task.	...
q8	The concepts of the modelling language were detailed enough to model the requirements.	
q9	I had difficulties in modelling user preferences with contributions to softgoals.	
q10	The effort of modelling seems too high for an efficient use in practice.	
q11	The obtained model is concrete enough to guide the programmers to an implementation respecting the requirements.	
q12	The obtained model is too abstract to be able to properly guide the programmers to an implementation respecting the requirements.	
q16	Enriching *Tropos* with conditions modelling seems useful for the scope of modelling adaptivity to the environment (*Tropos4AS* models only)	
q17	Enriching *Tropos* with failure modelling seems useful for the scope of modelling adaptivity to the environment (*Tropos4AS* models only)	
postq43	It was difficult to model the example, with all its details, with *Tropos*	
postq44	It was difficult to model the example, with all its details, with *Tropos4AS*	
postq45	In my opinion, the *Tropos* model captures the described requirements in a satisfying and complete way	
postq46	In my opinion, the *Tropos4AS* model captures the described requirements in a satisfying and complete way	
postq47	I had no difficulties in using the *Tropos4AS* extensions.	
postq48	I feel to have used the full potential of the *Tropos4AS* modelling concepts	
postq49	I think it is useful to have the extensions introduced with *Tropos4AS* for modelling requirements of systems that have to adapt to their environment.	
postq53	In my opinion, it is worth putting effort in modelling details of the requirements with *Tropos4AS*.	

Table 8.2: A selection of the questions in the questionnaires, with answers on a 1 . . . 5 Likert scale (on top of the table).

Finally, the post-questionnaire, [postq43]. . . [postq53] at the bottom of Table 8.2, collects data on a comparison of the two treatments *Tropos* and *Tropos4AS*, and on the usefulness of the *Tropos4AS* extensions.

Questions for both experiments:	
t	Time used for the task, in minutes
q1	Time spent for re-reading the modelling language tutorial in %
q2	Time spent for reading & understanding the example in %
q3	Time spent for modelling the example in %
Questions for the Tropos4AS experiment only:	
q13	Time spent for goal modelling in %
q14	Time spent for environment & conditions modelling in %
q15	Time spent for failure modelling in %

Table 8.3: Questions on the time spent on the activities in the experiment.

8.4 Data Analysis

For the analysis of the main factors, we (1) mapped the aspects $a1$ to $a7$, identified in Section 8.3.1, to one or more answers in the questionnaire, (2) applied a statistical analysis to evaluate the null-hypothesis expressed for each aspect, and (3) we grouped the aspects to answer to the two research questions $RQ1$ and $RQ2$.

The mapping between each aspect and the questions (reported in square brackets), can be found in the Tables 8.5, 8.6, and 8.7). To evaluate the main factor (i.e. to compare the two methodologies *Tropos* and *Tropos4AS*), the questions which were repeated for both treatments ([q1]. . . [q12]) can be directly compared with each other. Similarly, the answers in the post-questionnaire that are related, separately, to *Tropos* and to *Tropos4AS*, can be compared with one another (postq43/44 and postq45/46). The remaining answers, which are not directly related to any treatment or refer only to *Tropos4AS*, will be compared with the value 3, i.e., the *neutral* answer in the Likert scale used. The answers to the questions [q1], [q2] and [q3], capturing the relative time spent on the reading and modelling activities in %, are multiplied with the overall time, to obtain absolute time measures that can be compared between the two treatments.

8.4.1 Statistical evaluation

Proper statistical tests were selected to test if it is possible to reject the null-hypotheses or not (in this case, no conclusion can be drawn neither on rejection nor on acceptance of the hypothesis), based on the data obtained in the experimental tasks.

Considering the nature of the variables (they are not necessarily normally distributed), the limited number of data points (for most variables, two for each object), and the design of the experiment (balanced, with both treatments applied to each subject), we selected the non-parametric paired **Wilcoxon test** [Dalgaard, 2008] for evaluating the main factor (*Tropos* versus *Tropos4AS*). Since we have paired measurements for each subject, this test, which is based on a ranking of the differences in the pairs, can be used. Moreover, the Wilcoxon test does not make any assumption on the distribution in the sample.

We adopt a 5% significance level for the obtained ***p-values*** to determine if the null-hypothesis can be rejected or not. The p-value denotes the lowest possible significance with which it is possible to reject the null-hypothesis [Wohlin et al., 2000]. Hence, we will reject a null-hypothesis only if for the probability that it is true, p, we have $p < 0.05$.

Furthermore, for each data set we compute average ($\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$), median (the numeric value separating the higher half of a sample from the lower half) and the **Cohen.d effect size** (for a paired design: $d = \frac{\mu_2 - \mu_1}{\sigma_D}$, where σ_D is the standard deviation of the pairwise differences) to analyse trends and to estimate the magnitude of the obtained result, to facilitate its interpretation (as defined by Cohen [Cohen, 2004], 0.2: small, 0.5: medium, 0.8: large effect).

Co-factors are all factors different from the main factor (i.e. the two treatments), that could possibly have an (undesired) impact on the results of the experiment. An analysis to test if various co-factors (object, subject experience, subject position) had a statistically significant impact on the results, is made by a two-way **ANOVA** (Analysis of Variance) test. The calculation of ANOVA bases on the idea that a test for significance between means of different groups can be performed by comparing the two variance estimates [Dalgaard, 2008]. If the obtained p-values are smaller than 0.05, we have to reject our hypothesis that there was no relevant impact of a co-factor to the experiment results.

The statistical analysis is carried out with the *R* tool[5].

[5] *R* is a language and environment for statistical computing and graphics. *R* is a GNU project published under GNU General Public License and can be obtained at `http://www.r-project.org`.

8.4.2 Evaluation of model correctness

The evaluation of model correctness, for aspect $a8$, is carried our by an expert that analyses each model and identifies the plans that would be activated and the goals that fail in a certain scenario. Two scenarios were defined by the expert, for each of the two systems, the first one ($MQ1$) characterising some normal system behaviour, the second one ($MQ2$) including some exceptional happenings, see Table 8.4. Each scenario contains 3 or 4 relevant concepts, which have to be covered by the subjects' models[6]. The analysis of the 24 models obtained from the experiments (48 analyses with a total of 180 concepts), counting how many concepts are covered correctly for each scenario. The results are subjectively to some extent because of the big variety of models and different modelling ideas.

Example WMM
1) The machine is fully filled with 4kg of delicate clothes and the "Energy saving" wheel is in position 1 (most saving). Which plans will be activated? Expected result: Put 2 doses of detergent, set heating to 50°C, recycle water internally (3 concepts).
2) The machine is filled with 1kg of very dirty clothes and the "delicate" knob is pressed. The "Super Cleanness" wheel is in position 1 (most clean) and the "Energy saving" wheel is in position 0 (less saving). Expected result: Put 1 dose of detergent, set heating to 60°C, drain out dirt water and refill with fresh water, inhibiting heating (4 concepts).
Example PMA
3) The patient is very accurate and does not miss any meal and medicine. Which plans are activated during a day? Expected result: Monitor breakfast eating, monitor lunch eating, monitor dinner eating, monitor medicine intake (4 concepts).
4) The patient did not eat for breakfast. Now its time for lunch. Which plans will be executed? Suppose the patient will finally not eat for lunch, also after reminding him. Expected result: Monitor lunch eating, remind for lunch, "Eat lunch" fails and "Eat at least 2 meals" fails, so call care assistant (4 concepts).

Table 8.4: The four scenarios used to validate model correctness (in brackets, the number of concepts that are present in the scenario).

[6]Scenario 1 contains only 3 concepts, thus the result of the analysis is multiplied with $\frac{4}{3}$ to be comparable with the others.

8.5 Results and Interpretation

8.5.1 Adequateness of the experimental settings

Before analysing the main factors of the experimental study, we analyse if the experimental settings were adequate. The questionnaire contained several questions to evaluate if the participants had difficulties with the modelling languages, if they worked under time pressure, and if the object descriptions were clear. Moreover, the subject's position and their experience with *Tropos* is asked, to be able to calculate the influence of these co-factors.

	Question	median	reject null-hyp?	p-value
q4	The explanation of example clear to me	2	**Y**	0.0024
q6	I had enough time for accomplishing the task	2	**Y**	0.00012
preq15	I understood the basic notions of Tropos modelling	2	**Y**	0.0014
preq16	The visual notation used in Tropos is clear	1.5	**Y**	0.00095
preq17	The visual notation used in Tropos4AS is clear	2	**Y**	0.0034
postq47	I had no difficulties in using the Tropos4AS extensions	2	**Y**	0.02
preq48	I feel to have used the full potential of the Tropos4AS modelling concepts	3	N	0.6

Table 8.5: Results of the statistical analysis (Wilcoxon) for the adequateness of the experimental settings: medians and p-values for rejecting the null-hypotheses.

For each of the questions in Table 8.5, regarding the adequateness of the settings, we define a null-hypothesis of this form: The answer to the question is not significantly more positive than *"not certain"* (the neutral answer), i.e. the corresponding value 3. In the results, in Table 8.5, it can be observed that the median of 6 out of 7 questions we investigated, is 1.5 or 2. We can also observe that, by the statistical analysis performed (Wilcoxon), except for question [postq48], for all questions in Table 8.5 the p-values are smaller than 0.05, and thus the relative null-hypotheses can be rejected. Thus we can say that, although the subject experience with *Tropos* is small on average (result for question [preq13]), the initial understanding of both modelling languages seems to have been adequate (questions [preq15], [preq16], and [preq17]). The subjects did also not reveal particular difficulties in using the modelling concepts introduced with *Tropos4AS* [postq47]. However, many of the subjects had the feeling that the novel modelling concepts could be potentially better used [postq48].

Also, the objects of the experiment were adequate, since they were well-understood by the subjects [q4]. Moreover, the objects were considered nearly equally difficult (the PMA system was considered more difficult to model by 5/8 of the subjects) - this is also proven by co-factor analysis, later in this section. Most subjects also took enough time for completing the experiment (note that there was no fixed time limit given). Therefore, we can claim that in overall the experimental settings were adequate.

8.5.2 Main factor: results and interpretation

Aspect	median Tropos	median Tr4AS	reject null-hyp?	p-value	Cohen-d effect size
a1 [time] in minutes	49.5	75	**Y**	0.018	0.83 (large)
a2(a) [q10]	3.5	3	N	0.6	0.18 (negligible)
a3(a) [q1] in minutes	3.65	9.35	**Y**	0.0046	0.87 (large)
a3(b) [q2] in minutes	24.6	27.3	N	0.4	0.21 (small)
a3(c) [q3] in minutes	21.3	23.25	N	0.47	0.29 (small)
a4(a) [postq43postq44]	3	3	N	0.88	0 (small)
a4(b) [q5]	3	3	N	0.42	0.27 (small)
a5(a) [postq45postq46]	4	2	**Y**	0.023	1.7 (large)
a5(b) [q8]	3	2.5	N	0.2	0.42 (small)
a6 [q11]	3	2	**Y**	0.03	0.62 (medium)
a6 [q12]	4	2	**Y**	0.0039	0.68 (medium)
Model correctness:					
a8	3	4	**Y**	0.005	0.7 (medium)

Table 8.6: Results of the statistical analysis (comparison of Tropos vs. Tropos4AS) with Wilcoxon and rejection of the null-hypothesis. Related questions in square brackets.

We now give results for the main factor (the approach used), comparing the two treatments. The results of the statistical analysis are shown in Table 8.6 (for the results obtained by a comparison of values available for both modelling languages), and in Table 8.7 for the results of the questions regarding only *Tropos4AS*, compared with the *neutral* response (3 in the $1 \ldots 5$ Likert scale). Figure 8.2 reports boxplots for

	Median	reject null-hyp?	p-value	Cohen-d effect size
a2(b) [postq53]	2	**Y**	0.0043	1.16 (large)
a7(a) [q16]	2	**Y**	0.002	1.6 (large)
a7(b) [q17]	2	**Y**	0.006	1.04 (large)
a7(c) [postq49]	2	**Y**	0.001	2.1 (large)

Table 8.7: Results of the statistical analysis (values for Tropos4AS only) with Wilcoxon and rejection of the null-hypothesis

some of the aspects, giving a picture of the median and distribution of the results.

In the following, we interpret the results for each of the 8 aspects (a1,. . . ,a4 and a5,. . . ,a8), which were identified in Section 8.3.1, to detail the two research questions *RQ*1 and *RQ*2. Finally, the results are aggregated to give an answer to these research questions.

Research question 1: effort

(a1) Modelling an example with ***Tropos4AS*** **requires more time** than modelling it with *Tropos* (75 minutes vs. 50 minutes for the median, 51 vs. 72 for the average). With a p-value of 0.018 we can reject the null-hypothesis H_0a1, since we adopted a significance level of 0.05.

(a2) However, the **subjects perceive that the effort of modelling with *Tropos4AS* is not higher** than with *Tropos* ([q10]). This fact cannot be proven statistically (we cannot reject H_0a2), but the medians (3 vs. 3.5) show a trend that the effort is even slightly lower. We can speculate that giving the possibility for e.g. an explicit modelling of conditions and of the exceptional workflow enables to express the requirements in a more intuitive way, and thus decreases the perceived modelling effort. Moreover, the subjects agree that it is worth to put additional effort in modelling details of the requirements with *Tropos4AS*, with a median of 2 and statistical significance (data in Table 8.7) ([postq53]).

(a3) *Tropos4AS* requires **more effort for reading the language specification (question [q1]), while no statistically significant difference exists for the other activities** ([q2] and [q3]). This result is also confirmed by the medians and the Cohen-d effect sizes: for reading and understanding the example, and for modelling it, the time difference is negligible (11% in reading: median of 24.6 minutes for *Tropos* vs. 27.3 for *Tropos4AS*; 3% in modelling: 21.3 minutes for *Tropos* vs. 23.25 for *Tropos4AS*). The time averages, reported graphically in Figure 8.1, show the same trend (a huge difference in reading the language specification, but limited differences for requirements reading and modelling), albeit not to the same extend, because the arithmetic average is sensitive with respect to outliers. However, with two negative evaluations we cannot reject H_0a3.

(a4) *Tropos4AS* gives not more difficulties than *Tropos* for modelling the requirements of the object. This result cannot be confirmed statistically, by a comparison of [postq43] (regarding *Tropos*) and [postq44] (regarding *Tropos4AS*), but we obtained identical medians and averages, and thus also Cohen's effect size is zero. The subjects perceived the same or even slightly less difficulty in using *Tropos4AS* than *Tropos* [q5]. This result can be explained taking into account further comments of the participants: the additional modelling concepts introduced with *Tropos4AS* seem not only to bring higher complexity, but also to facilitate expressing the modelling intentions. Thus, we cannot reject H_0a4.

Considering all these four aspects, we have to answer in affirmative way to the research question $RQ1$:

> *Yes, the effort required to apply Tropos4AS is higher than the effort required to apply Tropos. However, the additional effort is not perceived by the users as such. They do not face particular difficulties, and spent significantly more time only for studying the new Tropos4AS modelling concepts.*

Research question 2: effectiveness

(a5) The participants agree that *Tropos4AS* produces models more expressive than *Tropos* models [postq45postq46]. Moreover, although, the relative null-hypothesis H_0a5 cannot be rejected, the medians and the effect size show a trend that the participants were more confident for *Tropos4AS*, than for *Tropos*, that the concepts of the modelling language are detailed enough for modelling the requirements [q8].

(a6) *Tropos4AS* is perceived to be more effective than *Tropos*, for producing models that are concrete and can guide the developers to the implementation [q11, q12]. H_0a6 can be rejected with statistical significance, with p-values of 0.03 for [q11] and 0.0039 for [q12].

(a7) The subjects agree that enriching *Tropos* is useful for modelling adaptivity (in general, and also for both the conditions and the failures) [postq49,q16,q17]. H_0a7 can be rejected.

(a8) The analysis of model correctness performed evaluating the relative scenarios (Table 8.4) shows with statistical evidence (i.e., with a p-value of only 0.005, H_0a8 can be rejected), that the models produced with *Tropos4AS* are more correct than the models produced only with *Tropos*. The medians, 4 of 4 for *Tropos4AS* and 3 of 4 for *Tropos*, and the averages (2.7 for *Tropos* and 3.5 for *Tropos4AS*) confirm this result.

Considering all these four aspects, we can answer in an affirmative way to the research question $RQ2$:

> *Yes, Tropos4AS allows the users to produce models more effective than Tropos for representing requirements of an adaptive system.*

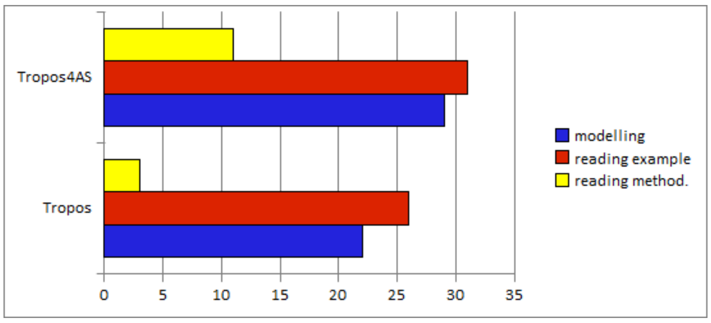


Figure 8.1: Comparison of the averages of times spent in the different activities: reading the language specification, reading the requirements and modelling.

8.5.3 Additional results

An additional analysis of the results shows various findings that are not directly related to the research questions, but are important for understanding how *Tropos* and *Tropos4AS* are used in practice, by both experienced and novice software engineers.

The participants spent from 27 to 90 minutes for the *Tropos* assignment, and from 37 minutes to 110 minutes for *Tropos4AS*, with high, but very similar variances. More details can be seen in the boxplots in Figure 8.2, which give an immediate graphi-

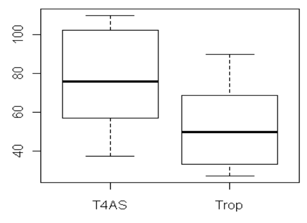
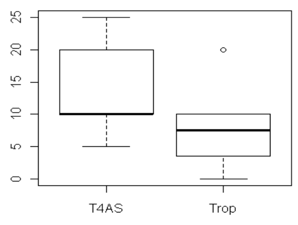
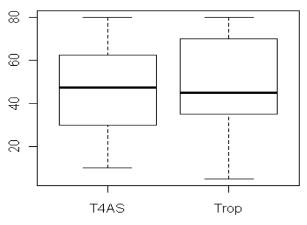
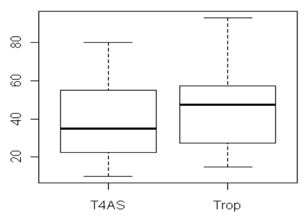
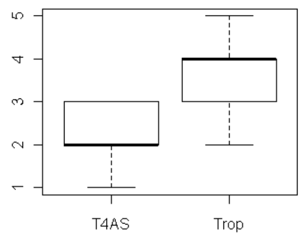
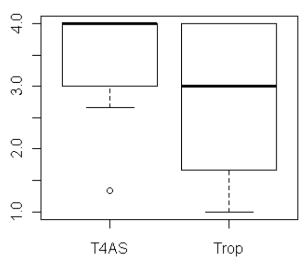
Time overall 	Time read method % [q1] 	Time read example % [q2] 
Time modelling % [q3] 	[postq45] vs. [postq46] 	Correctness (evaluated) 

Figure 8.2: Boxplots with median, lower and upper quartiles, minimum, maximum, and eventual outliers, for selected questions. Differences in the range of values, in their distribution and medians become visible.

cal impression of the differences. They show the median, upper and lower quartile[7], minimum, maximum, and eventual outliers, for selected results. On average, for both treatments together, the participants spent 47% of their time with the reading of requirements and only 42% with modelling. 11% were used for looking at the language specification, but this time could for sure be reduced with a better training. The times for *Tropos* and *Tropos4AS* can be compared with the help of Figure 8.1. Moreover, creating *Tropos4AS* models, on average 59% of the time was used for goal modelling, 25% for conditions modelling and only 16% for failure modelling.

The subjects experienced only low difficulties in using the new extensions, but were not certain to have used the full potential of them. In their opinion it is worth putting effort in modelling such details, for supporting the developers. For both treatments, they had some difficulty in modelling softgoal contributions (median 3), but they agreed

[7]Quartile: the region containing, respectively, the lower 50% of values higher than the average, and the upper 50% of values lower than the average

on the usefulness of goal decomposition (median 2 on the 1...5 Likert scale).

The subjects used most extensions as expected, accordingly to the modelling philosophy. They only had some difficulty with the semantics of the different types of condition and goals. This was expected due to the brief training and the missing knowledge about the implementation. However, the correctness of the models created with *Tropos4AS* is significantly higher (with a median of 4, the maximum value) compared to the *Tropos* models. With one exception, the models covered the evaluation scenarios by more than 66%, as it can be seen in the bottom right boxplot of Figure 8.2.

8.5.4 Co-factors

We identified three relevant co-factors and analysed if they could have an undesired impact on the experiment: the position of the subject (joined in two groups: researchers + post-docs and students + programmers), the experience in working with *Tropos* (low or high) and the objects of the experiment (WMM and PMA). By an ANOVA (analysis of variance) test, we analyse the presence of statistically significant (but undesired) interaction between a co-factor and the main factor (i.e. the two treatments). Moreover, the ANOVA test also reports (with a p-value) if there are statistically significant differences in the test results, partitioning them not by treatment, but by one of the co-factors.

The three co-factors seem to have had only a very limited impact on the experiment. In Table 8.8 we report the co-factor analysis relative to the experience of the subjects in working with *Tropos*.

Observing the p-values in the third row of Table 8.8, we can see that the subjects which indicated to be experienced with *Tropos*, (notice that this set is nearly congruent to the set of researchers and post-docs), produce (in general, despite the treatment) more correct models than students and programmers (aspect a8). Moreover, this set of subjects perceives less effort in modelling (aspect a2) and gives stronger agreement to the claim that the concepts of the modelling languages are detailed enough to model the requirements (aspects a5 and a6). However, despite these results, no statistically significant impact can be seen, considering the interaction between the experience and the treatment, i.e. the p-values in row 4 of Table 8.8. This can be explained by the random, even distribution of the experienced participants in the experiment groups.

An analysis of the other co-factors gave similar, but statistically less significant results. Only one variable, for aspect (a4b), gave a significant difference in the interaction between object and treatment (p-value = 0.03). This means that the participants had slightly more difficulty in modelling the personal monitoring agent (PMA) example.

Aspect	Treatment	Experience	Treatment:Experience
a1 [time]	0.42	0.75	0.52
a2(a) [q10]	0.34	**0.0049**	0.74
a2(b) [postq53]	x	0.89	x
a3(a) [q1m]	**0.003**	0.14	0.91
a3(b) [q2m]	0.44	0.69	0.7
a3(c) [q3m]	0.32	0.71	0.41
a4(a) [postq43postq44]	1	0.12	0.67
a4(b) [q5]	0.35	0.07	0.74
a5(a) [postq45postq46]	**0.0005**	0.58	0.052
a5(b) [q8]	0.14	**0.02**	0.21
a6 [q12]	**0.0061**	**0.024**	0.17
a6 [q11]	**0.02**	0.3	0.69
a7(a) [q16]	x	0.63	x
a7(b) [q17]	x	0.56	x
a7(c) [postq49]	x	0.65	x
a8	**0.0027**	**0.04**	0.91

Table 8.8: Co-factor analysis (ANOVA) for the 8 aspects, for the interaction between the treatment and the subject's experience. The table reports the p-values obtained by an analysis regarding the treatment (these values are expected to have the same trend as the results of the Wilcoxon test), regarding the subject's experience only, and regarding interaction between the treatment and the experience. Aspect a7 contains only values for *Tropos4AS*.

8.5.5 Threats to validity

Conclusion validity The experiment has a balanced design, with both treatments applied to each subject, to lower the effect of the different experience of the participants. Proper statistical tests were performed to reject the null-hypotheses. We used the non-parametric Wilcoxon test and ANOVA (which is quite robust with respect to the non-normality of the distribution of the samples) and no specific assumptions on the distribution of the samples. The medians and Cohen's effect size confirm the results. We make available the anonymised raw data of the experiment online (`http://selab.fbk.eu/morandini/T4ASmodellingexperiment_rawdata.zip`).

Internal validity We adopted a paired and balanced design to mitigate learning effects and group assignment. The number of tests (24, two for each subjects) is reasonably high to get statistical validity. The subjects had different background knowledge, but the division into groups was balanced. This is confirmed by an analysis of various co-factors that did not give any suspected result that would reduce internal validity. Also, the average times spent for each treatment and the correctness of the results showed that the subjects were still motivated in the second laboratory of the experiment.

Construct validity Half of the subjects did not have any reasonable knowledge of the *Tropos* methodology, but the differences in performance between *Tropos* and *Tropos4AS* modelling were similar to *Tropos* experts (proven by co-factor analysis). Moreover, the modelling languages were mostly used in a correct way, thus we can conclude that the short tutorial and the available documentation were adequate.

The experiment reflects the use of the modelling languages, not the modelling process of the methodologies. A big part of our measures results from subjects' opinions and gives thus a subjective impression of the object under study. Nevertheless we have an objective measure of the time for $RQ1$, and $RQ2$ is supported by a measure of the model correctness, performed by an expert.

For achieving completely objective results, we would have had to limit ourselves to a small piece of modelling, or to a comprehension task. Both of these are not suitable for the aim of understanding how software engineers would accept and use the newly proposed extensions. The objects under study (i.e. the systems to model) are small, even if not trivial. In fact they required a quite high understanding and modelling effort. Much more complex objects would not be treatable in such an empirical study. Performing an industrial case study was not considered for this primary evaluation of a new methodology. However, such a study would give complementary results to an empirical study with subjects.

External validity All subjects are part of a single research group, but most of them are not working in requirements engineering or modelling. Because of similar results achieved with expert and non-expert participants, we expect that these results can be generalised to subjects without a deep background on *Tropos* (e.g. bachelor or master students) or to experts in goal modelling. Only further studies, explicitly focussing on these differences (e.g. as in [Ricca et al., 2010]) could evaluate this. It remains doubtful if professional, traditional engineers would give similar results for the effectiveness of both the methodologies. This would be a threat to *Tropos* in general and not specifically to the extensions in test.

However, our evaluation is not intended to get feedback on how subjects perform with *Tropos*, but on how performance is improved with the extensions introduced. To encourage a repetition of the study with different subjects, the replication package, containing the instructions and the work packages for the subjects, is available online at `http://selab.fbk.eu/morandini/T4ASmodellingexperiment_package.zip`,

8.6 Experiment 2: Comprehension

Having previously examined the effort of modelling and the effectiveness of the models created, the goal of the second part of this empirical study is to examine the comprehensibility of models created by applying the two methodologies.

The experiment, lasting about thirty minutes in total, consists of several comprehension questions, which have to be answered by the subjects in a concise time, using both the model and the textual requirements specification, and additional questions to strengthen the results of the analysis. It is carried out with the same subjects, objects and paired, counterbalanced design. The participants were reassigned randomly to the groups in Table 8.1.

8.6.1 Research questions and hypotheses

With the goal of evaluating the comprehensibility of *Tropos4AS* models in comparison to traditional *Tropos* models, we define the following research question:

*RQ*3: Do *Tropos4AS* models significantly improve the comprehension of the requirements of a self-adaptive system, in comparison to *Tropos* models?

The research question has been translated, with the expected direction (they are one-tailed), to the corresponding *null-hypothesis* $H_0 3$ and *alternative hypothesis* $H_a 3$:

- $H_0 3$: The comprehensibility of system requirements **cannot be** significantly improved by using *Tropos4AS* models, in comparison to *Tropos* models.

- H_a3: The comprehensibility of system requirements **can be** significantly improved by using *Tropos4AS* models, in comparison to *Tropos* models.

8.6.2 Experiment design and discussion

To evaluate the comprehensibility of the two modelling languages, we set up an experiment to measure the subjects' effectiveness of retrieving correct information from a model. However, we have to consider that *Tropos4AS* includes various additional concepts to *Tropos*. Thus, the resulting models are inevitably more complex. However, therefore, as also shown in the previous part of the experiment, they are more expressive and contain more details of the requirements than *Tropos* models.

To achieve a fair comparison of the comprehensibility, the same amount of details must be present for both treatments. Basing the comparison only on models, this would be problematic in our case, since *Tropos4AS* is a specific *extension* of *Tropos*. Thus, we provide the models together with a textual summary of the systems' requirements specifications. This represents a reasonably realistic scenario in which a modeller uses both the models and the specifications to understand a system.

The experiment design bases on the assumption that the comprehension tasks can be carried out faster by looking at the models than by reading the textual specification. We therefore give a strict time limit of 12 minutes for each laboratory, to encourage the use of the diagram and avoid that the tasks are resolved directly by reading the specifications. Performing a pilot study prior to the execution of the experiment (with different subjects), we recognised that 12 minutes were an appropriate time for our needs.

Since we also hand out the textual requirements specifications, we have also to evaluate our assumption, by asking the subjects for the amount of information extracted from the model and from the textual specifications. Hence, research question $RQ3$ will be covered by the following questions:

$RQ3_1$: Can the effectiveness of retrieving correct information (in limited time) be increased by combining the textual specifications with *Tropos4AS* models, than when combining them with *Tropos* models?

$RQ3_2$: Is a *Tropos4AS* model more useful than the *Tropos* model, to extract the information asked?

To evaluate this question in our experimental setup, we analyse the following aspects a, b and c. For the sake of conciseness, we do not report here the null- and alternate hypotheses for each aspect.

a) the correctness of the information extracted by the subjects from the models and the textual specifications

b) the amount of information extracted from the model

c) the amount of information extracted from the textual requirements specification

Objects For this second experiment, we have deliberatively chosen to use the same objects as in the first experiments, that is, the requirements of the systems **WMM** and **PMA**. Executing the experiment about three months after the first one, the participants will remember the context, but no details.

8.6.3 Experiment procedure and material

The experiment consists of a short introduction, and two laboratories lasting exactly 12 minutes, each one followed by a short questionnaire with four questions. The laboratories are done in small groups, in a time frame of approximately 30 minutes.

Prepared input material To perform the experiment, each participant receives the following material, for each laboratory (a complete sample is shown in Appendix A, page 235:

- A summary of the modelling language to use (either *Tropos* or *Tropos4AS*).
- The requirements specification of the system (WMM or PMA).
- The model of the WMM or PMA system, constructed either with *Tropos* or *Tropos4AS*.
- 5 model comprehension questions, either for WMM or PMA.
- A short questionnaire.

The system models handed out to the subjects are of fundamental importance for the success of the comprehension experiment. To ensure a fair comparison, for creating them, we observed the following process.

We analysed the models obtained from the previous experiment and constructed both the *Tropos* and *Tropos4AS* models, taking ideas and model parts from models built by the participants during that experiment. We then corrected these models, to correctly capture a high number of requirements (hence obtaining a sort of "gold standard" model written by an expert), but still respecting the characteristics of the modelling language. The four resulting models, two for each WMM and PMA, can be looked up in Appendix A, page 237.

The model comprehension questions (Appendix A, page 239) are general questions on the requirements specifications. The main parts of the answers is included in the *Tropos4AS* and *Tropos* models. Remaining details can be found in the 1-page textual requirements specifications.

Experiment task The subjects should answer the model comprehension questions. This can be done by looking at the goal model (either *Tropos* or *Tropos4AS*) assigned to them, or, if the answers are not in the model, or the subjects encounter difficulties in extracting them, by searching them in the textual specifications. Each task has a strict time limit of 12 minutes. After it, the short questionnaire (Table 8.9) should be filled in.

cq1	I had enough time for accomplishing the tasks
	☐1 strongly agree ☐2 agree ☐3 not certain ☐4 disagree ☐5 strongly disagree
cq2	The comprehension questions were clear
	☐1 strongly agree ☐2 agree ☐3 not certain ☐4 disagree ☐5 strongly disagree
cq3	I was able to extract the asked information from the goal model
	☐1 nearly all ☐2 most ☐3 half ☐4 somewhat ☐5: nearly nothing
cq4	I needed to search the information in the textual requirements
	☐1 nearly all ☐2 most ☐3 half ☐4 somewhat ☐5: nearly nothing

Table 8.9: Comprehension experiment: questionnaire.

8.6.4 Variables and measures

The *independent variable* of the study is again the modelling language used, considering the treatments *Tropos* and *Tropos4AS*. The *dependent variables* are the correctness of the subjects' answers to the comprehension questions and the amount of information extracted by the subjects from models and textual requirements.

To measure the comprehension level and test the hypotheses, we assessed the answers to the comprehension questionnaire. Since we expected each answer in terms of a list of elements, we evaluated each answer of the subjects by measuring *precision*, *recall* and *f-measure*. In particular, considering $Answ_{s,i,t}$, the set of elements mentioned in the answer given with treatment t by a subject s in question i, and $ExpAnsw_i$, the set of elements we expected to be in the correct answer for question i, we measured precision, recall and f-measure, defined as follows:

$$Precision_{s,i,t} = \frac{|Answ_{s,i,t} \cap ExpAnsw_i|}{|Answ_{s,i,t}|}$$

$$Recall_{s,i,t} = \frac{|Answ_{s,i,t} \cap ExpAnsw_i|}{|ExpAnsw_i|}$$

$$F - measure_{s,i,t} = \frac{2 \cdot Precision_{s,i,t} \cdot Recall_{s,i,t}}{Precision_{s,i,t} + Recall_{s,i,t}}$$

These three measures represent continuous variables in the range [0,1]. *Precision* indicates the fraction of correct elements out of the elements in the answer, whereas *recall* indicates the fraction of relevant elements, that were retrieved. The *F-measure* combines these two measures, by calculating their harmonic mean, into a single measure which represents the effectiveness of the answer retrieval. The average of each of these measures, by subject, for the 5 questions, has been used for the statistical analysis of the result, e.g.:

$$Avg(Precision)_{s,t} = \frac{\sum_{i=1}^{5} Precision_{s,i,t}}{5}$$

where $\{Avg(Precision)_{s,T1}\}$, for subjects $s = 1 \ldots n$ and treatment $T1$ (e.g. *Tropos*), is be the set of data in input to the statistical analysis.

We additionally asked the subjects 4 questions in a questionnaire. Two questions (cp3, cp4) concern the source (model and/or requirements specification) used by a subject to extract the information required to answer each question and are defined on a scale from 1:*nearly all* to 5:*nearly nothing* of the amount of information extracted from the models or the textual requirements (see Table 8.9). The questions cp1 and cp2 concern the adequateness of the experimental settings (i.e., the time to fill the questionnaire and the clearness of the questions) and use ordinal Likert scale variables, defined on a scale from 1 to 5, as follows: 1 strongly agree; 2 agree; 3 not certain (neutral answer); 4 disagree; 5: strongly disagree.

Notice that we fixed the time to answer the questions, thus *time* is not considered as a depended variable in this experiment. Possible co-factors of the experiment might be the object, the subject experience and position, and the laboratory.

Remark Precision, recall and f-measure cannot be defined if the respective denominator is 0. This could happen in two cases: (1) the set $ExpAnsw_i$ is empty (i.e., no correct answer exists for question i); or (2) the set $Answ_{s,i}$ is empty (i.e., the subject s gives no answer to question i). In our case, $ExpAnsw_i$ is never empty, thus only the second case could happen, making the precision (for this specific subject and question) undefined in some cases.

In these cases, we decided to set $precision_{s,i} = 0$, with the aim of preserving the meaning of the precision measure and also, at the same time, taking into account the not given answer. However, as a double-check, we repeated the analysis using precision, recall and f-measure computed considering the entire set of answers given by each subject (i.e., we put together all answers given by the subject s). In other terms, for each subject we compute $Precision_s$, $Recall_s$ and $F - measure_s$ using: $Answ_s$, the set of the sets of elements in the answers given by a subject s; and $ExpAns_s$, the set of the sets of correct answers. This approach to compute precision, recall and f-measure is often used in literature (e.g., [Bacchelli et al., 2010, Antoniol et al., 2002]) to overcome the problem of having "undefined" measures for precision and recall. In our case, however, the main disadvantage of this second approach is, that having different sets of elements in the correct answer expected for each question, we could not give different weights to the answers given by the subjects.

Hence, considering that we have few cases in which subjects cannot answer to our questionnaire, we prefer to focus on the first computation method for precision, recall and f-measure, and using the second method only to double-check the results.

8.6.5 Statistical evaluation

To analyse the results with respect to the main factor – considering the nature of the variables (Precision, recall and f-measure are continuous in the range [0,1]), the limited number of data points, and the paired experimental design – we use the paired, non-parametric *Wilcoxon* test, adopting a 5% significance level. Where useful, we use the *Cohen.d effect size* to estimate the magnitude of the results obtained (see Section 8.4.1 for details).

To analyse the four questions cq1-cq4 of the questionnaire we adopted the following strategy: Considering the nature of variables (1-5 on an ordinal scale) and the limited number of data points, for each of the questions we applied the paired non-parametric Wilcoxon test, adopting a 5% significance level, in two analyses:

(A) Comparison of the whole set of answers with the threshold 3, representing the neutral answer in the Likert scale used.

(B) Comparison of the answers given by the use of *Tropos4AS* with those given by the use of *Tropos*.

Additionally, for better studying cq3 and cq4, we performed a further analysis:

(C) We subdivided the answers into two groups: $G1$ (answers < 3) and $G2$ (answers > 3). The answers equal to 3 have been added to $G1$ for cq3 and to $G2$ for cq4, according to the expected trend. Then, we counted the answers of each group, $G1$ and $G2$, relative to the two treatments (*Tropos* vs. *Tropos4AS*).

Analysis (C) has additionally been applied considering both the answers to cq3 and cq4, grouping in $G1$ all the subjects having answers < 3 in cq3 and at the same time answers > 3 in cq4. The remaining set of subjects is part of the group $G2$.

To conclude the experiment, we applied a two-way ANOVA (analysis of variance) test to analyse if there is a statistically significant impact of various co-factors (the laboratory, the objects, subject experience and position) on the results. Results are reported in Section 8.7.3.

8.7 Results and Interpretation

8.7.1 Adequateness of the experimental settings

Before analysing the main factors, we analyse if the experimental settings were adequate. The questionnaire contained two questions cq1 and cq2 to evaluate if the participants worked under time pressure, and if the example descriptions were clear (ref. to Table 8.9). The associated high-level research questions (RQ_{cq1}: *Is the time to fill the questionnaire adequate?* and RQ_{cq2}: *Is the clearness of the questions adequate?*) result to the following Hypotheses:

- Null-hypothesis H_0cq1: The time to fill the questionnaire is not adequate.
- Alternate hypothesis H_acq1: The time to fill the questionnaire is adequate.
- Null-hypothesis H_0cq2: The clearness of the questions is not adequate.
- Alternate hypothesis H_acq2: The clearness of the questions is adequate.

Question (see Table 8.9)	median	reject null-hyp?	p-value
cq1 adequateness of time	2	Y	0.049
cq2 clearness of questions	2	Y	0.000015

Table 8.10: Results of the statistical analysis (Wilcoxon) for the adequateness of the experimental settings.

We can define the answers "strongly agree" and "agree" (1 and 2 on the Likert scale used) as *adequate* , comparing the answers with a threshold of 3 (the neutral answer). The p-values (in Table 8.10) obtained by applying the Wilcoxon test, lead us to the following comments:

The hypothesis H_0cq2 can be rejected, with a very small p-value of $1.5 \cdot 10^{-5}$, thus we can state that the comprehension questions have been perceived to be clear. The time has been perceived as adequate (p-value 0.049), even if the subjects have the feeling that more time was better for answering the questions. However, a tight time limit was selected on purpose, to avoid that the participants are able to read the whole textual specifications.

Also, the subjects perceived a bit less time pressure when using *Tropos4AS* (median 2) than *Tropos* (median 2.5), but these differences are not statistically significant. Overall, we can confirm that the experiment setting have been perceived as adequate by the subjects.

8.7.2 Main factor: results and interpretation

To give an answer to the research question $RQ3$ – understanding if *Tropos4AS* models improve the comprehension of the requirements of a system – now we analyse the results of the statistical analysis.

The analysis of the answers to the comprehension questions was repeated using both methods we defined in Section 8.6.4, to compute precision, recall and f-measure. However, considering that: (1) we obtained only one case (out of 120 received answers) in which a subject gave no answer for a question (out of the ten considered per subject); and (2) the results we obtained repeating the analysis with both methods are very similar (only minor differences exist), we decided to omit the description of the results obtained using the second method, and describe the results obtained by the first calculation method, considering the average of precision, recall and f-measure on all questions, for each subject. The second computation method confirms the results obtained. Figure 8.3 shows the boxplots of the obtained results for precision, recall, and f-measure.

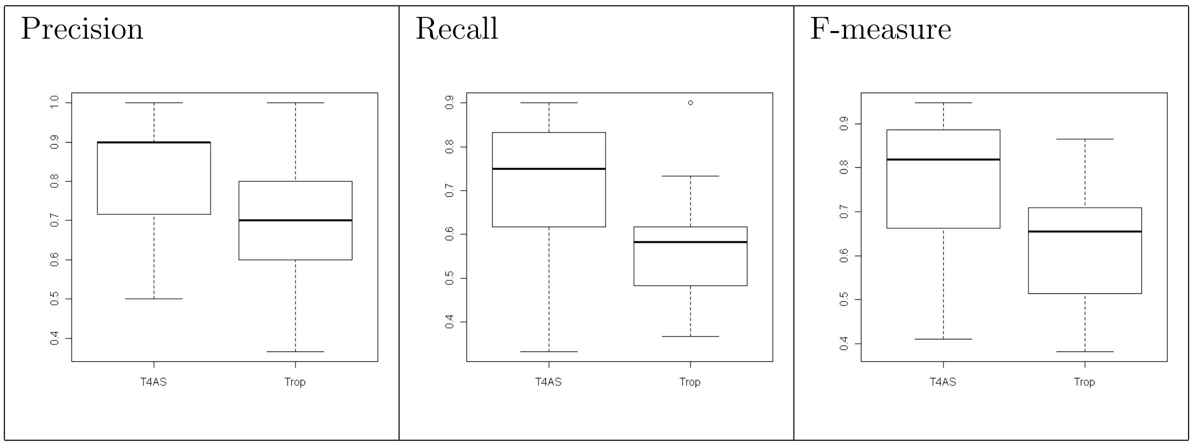


Figure 8.3: Boxplots for the distribution of the averages per subject, for precision, recall and f-measure of the single answers to the comprehension questions.

Aspect	median Tropos	median Tr4AS	reject null-hyp?	p-value	Cohen-d effect size
Precision	0.7	0.9	Y	0.021	0.74 (medium)
Recall	0.58	0.75	Y	0.020	0.82 (large)
F-measure	0.65	0.81	Y	0.008	0.87 (large)

Table 8.11: Comprehension test: results of the statistical analysis for the main factor, with a paired Wilcoxon test.

By looking at Table 8.11, we observe that *Tropos4AS* significantly improved the precision of the subject answers (referring to the median, form 70% to 90%). This result is statistically relevant and supported by a medium effect size. In other terms, from a *Tropos4AS* model (together with the textual requirements specifications), in a fixed, limited time, "more correct/precise" information can be extracted, in comparison to a *Tropos* model (together with the same specifications).

Tropos4AS also improved the recall of the subjects' answers with statistically significant evidence (form 58% to 75% for the median). Also this is a statistically supported result. In other terms, information extracted from a *Tropos4AS* model can be expected to be "more complete" with respect to those extracted from a *Tropos* model.

Correspondingly, *Tropos4AS* improved, with statistical significance, also the f-measure of the subject answers (form 65% to 81% for the median), denoting the general effectiveness of information retrieval.

Therefore, we can answer with statistical evidence in an affirmative way to the research question $RQ3_1$:

> *Yes, a Tropos4AS model, together with the textual specifications, is more effective for retrieving correct information than a Tropos model, together with the same textual specifications.*

To understand if the *Tropos4AS* model was more useful to extract the information asked, than the *Tropos* model (research question $R3_2$), we analyse the answers given to cq3 and to cq4 and moreover also to cq3 and cq4 together, as defined in Section 8.6.5 (B) and (C).

The *Tropos4AS* models seem to contain information more useful to answer the questions. In particular, the results of the analysis (analysis B in Section 8.6.5), shown in Table 8.12, show for question cq3 that the amount of information extracted from *Tropos4AS* models is (quantitatively) larger than the amount extracted from *Tropos*. Similarly, the results for question cq4 confirm that the amount of the information extracted from textual requirements is (quantitatively) lower when using *Tropos4AS* than when using *Tropos* models. These results are confirmed with statistical significance (p-value $< 5\%$) and a large effect size. Figure 8.4 shows this distribution graphically. A median of 1 ("nearly all") for the information extracted from the *Tropos4AS* model, in comparison to a median of 3 ("half") for *Tropos*, shows that not only the *Tropos4AS* answers are more correct (RQ_1), but also that they were retrieved to a much higher amount from the models. The use of the textual requirements specifications (qc4) by the subjects seems to be quite limited in general, with a median of 4 for *Tropos4AS* and *Tropos* together. Comparing *Tropos4AS* models and *Tropos* models, our hypothesis that the subjects were able to extract most of the information from the *Tropos4AS* model, is confirmed (median=4.5), while using the *Tropos* model, half of the information, on average, had to be extracted from the textual specifications (median=3).

Aspect	median Tropos	median Tr4AS	reject null-hyp?	p-value	Cohen-d effect size	expected value
cq3 (model)	3	1	Y	0.015	0.8 (large)	low
cq4 (text)	3	4.5	Y	0.038	1.2 (large)	high

Table 8.12: Analysis (Wilcoxon) of the main factor, for the amount of information extracted from the model (cq3) and from the specifications (cq4).

To show more concrete results, we apply the analysis C, defined in Section 8.6.5. For both questions cq3 and cq4, we group the "positive" answers (from the viewpoint

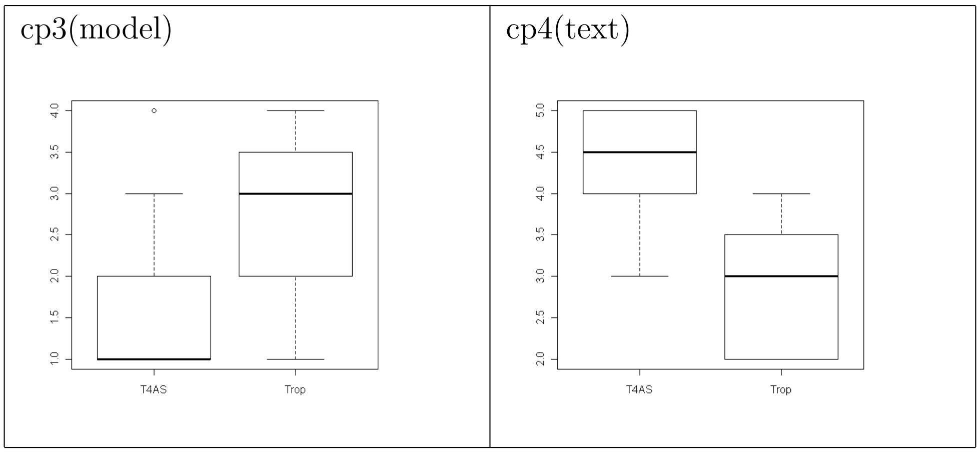


Figure 8.4: Boxplots comparing the main factor, for the questions cp3 (amount extracted from the model) and cp4 (amount extracted from the textual specifications. Y-axis: 1 (nearly all) to 5 (nearly nothing).

of the use of the model) to the respective sets $G1$ and $G2$. The results, shown in Table 8.13, confirm the previous ones. 4 out of 12 subjects (33.3%) found the *Tropos* models (almost) sufficient to answer the questions while 8 out of 12 (66.6%) were not able to extract from the models more than half of the information asked, while the *Tropos4AS* model was (almost) sufficient for 10 out of the 12 subjects (83.3%).

On the other side, 9 out of 12 subjects (75%) that used *Tropos* models, needed to extract a significant amount of the answers from the textual requirements specifications, while only 2 out of 12 subjects (16.6%) that used the *Tropos4AS* models needed them for a substantial part of the answers to the comprehension questions.

To ignore eventual false positives, in a last analysis we combine the questions cq3 and cq4. Only the subjects that gave a "positive" answer to both questions for a treatment, are grouped to the group $G1_+$, while all the rest is put to $G2_+$. In other words, the subjects in $G1_+$ found the models almost sufficient to answer the questions and used the requirements only partially.

The results in the lower part of Table 8.13 show that 3 out of 12 subjects (25%) found the *Tropos* models sufficient to answer the questions while using only partially the requirements. On the other hand, 10 out of 12 subjects (83.4%) found the *Tropos4AS* models sufficient to answer the questions and used only partially the requirements. Combining the results of the evaluation of the questions cq3 and cq4, we can affirm that in the experiment, the *Tropos4AS* model was more useful than the *Tropos* model, to extract the information asked ($RQ3_2$).

Summing up, we can answer with statistical evidence to the main research

cq3	$G1_3$ (answers<3)	$G2_3$ (answers $\geq$ 3)
Tropos	4	8
Tropos4AS	10	2
cq4	$G1_4$ (answers$\leq$ 3)	$G2_4$ (answers>3)
Tropos	9	3
Tropos4AS	2	10
cq3 & cq4	$G1_+$ (cq3: answers $<$ 3 $\wedge$ cq4: answers $>$ 3)	$G2_+$ (all others)
Tropos	3	9
Tropos4AS	10	2

Table 8.13: Count of subject answers, grouped around the threshold 3.

questions: *Tropos4AS* is more effective than *Tropos* to support retrieving correct information from the requirements specifications ($RQ3_1$); in this evaluation, a substantial part of the information was extracted from the models. Specifically, *Tropos4AS* models were more useful than *Tropos* models, to extract the information asked ($RQ3_2$).

Considering these results, we can reject the null-hypothesis H_03 and answer in an affirmative way to the research question $RQ3$:

> *Yes, the comprehensibility of system requirements can be significantly increased by using Tropos4AS models, in comparison to Tropos models.*

8.7.3 Co-factors

We investigated the impact of four main co-factors in the experiment: the laboratory (first vs. second), the position of the subject (researchers vs. PhD-students & programmers), the experience of the subjects in working with *Tropos* (low vs. high)[8], and the objects of the experiment (WMM vs. PMA). This analysis was concluded without observing any kind of statistically relevant impact on the main factor, with respect to the treatment, see the results in Table 8.14.

Only the experience of the subjects with *Tropos* shows a small, but not statistically confirmed impact, when considered with respect to the treatment. From the interac-

[8]Position and experience of each subject have been enquired in the first experiment.

	Treatment	Laboratory	Treatment:Laboratory
Precision	0.08	0.75	0.69
Recall	0.048	0.93	0.81
F-measure	0.044	0.82	0.71
	Treatment	Position	Treatment:Position
Precision	0.07	0.74	0.19
Recall	0.036	0.91	0.12
F-measure	0.035	0.98	0.14
	Treatment	Experience	Treatment:Experience
Precision	0.07	0.62	0.13
Recall	0.0.34	0.8	**0.08***
F-measure	0.031	0.67	**0.08***
	Treatment	Object	Treatment:Object
Precision	0.071	0.15	0.59
Recall	0.037	0.16	0.48
F-measure	0.033	0.13	0.46

Table 8.14: Co-factor analysis (ANOVA) for the interaction between the main factor (the treatment) and the co-factors.

tion plots (Figure 8.5) showing the interaction between treatment and experience with respect to precision, recall, and f-measure, we can make interesting observations:

– Subjects with high experience achieved a performance not particularly influenced by the treatment.
– Instead, subjects with low experience achieved a performance quite influenced by the treatment, in fact, their results have been improved with *Tropos4AS* by about 20/25%.
– An interaction between experience and treatment exists, even if it is not statistically relevant.
– The difference of the performance of subjects with low and high experience seems to be increased when considering *Tropos4AS*. Moreover, with the *Tropos4AS* treatment, subjects with low experience overcame (about +10%) the results obtained by subjects with high experience.

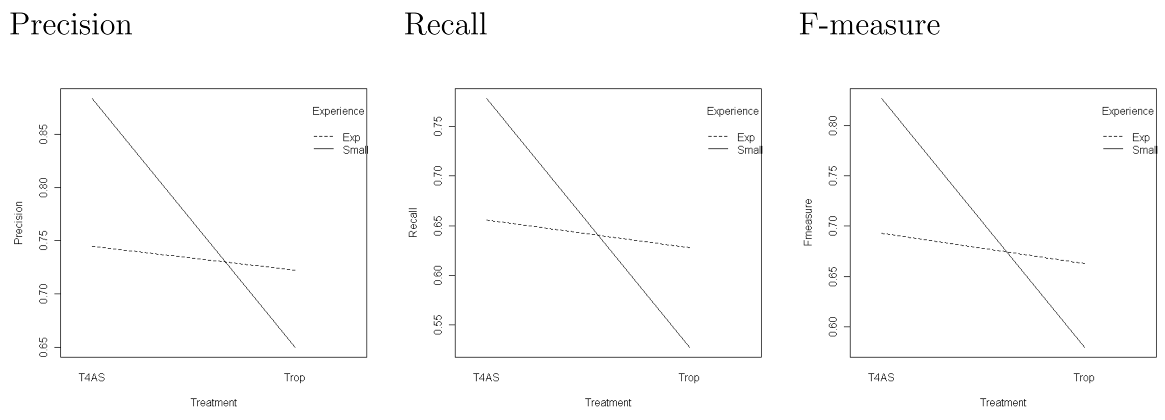


Figure 8.5: Interaction plots for the treatment with respect to the subject's experience (experienced vs. few/small experience), for Precision, Recall and F-Measure.

8.7.4 Threats to validity

Having used the same balanced experimental design, identical subjects and the same statistical evaluation methods, the considerations made in Section 8.5.5 are valid also for the comprehension experiment: A paired, balanced design was adopted to mitigate learning effects. The subjects remained unchanged and the co-factor analysis has shown no relevant differences in the performance, with respect to position, expertise on *Tropos* and the objects of the experiment. We make available the anonymised raw data of the experiment online at `http://selab.fbk.eu/morandini/T4AScomprehensionexperiment_rawdata.zip`.

By construction of the experimental task, additional possible threats to validity come up. The evaluation was not based on the subjects' perceptions and opinions, but on more objective questions. However, two parts of the experimental setup could have influenced the results: the models provided for the comprehension questions, and the questions themselves. As already mentioned, we constructed the models by meticulously analysing the models constructed by the participants of the first experiment, and by paying attention to the peculiarities of the methodologies. Thus, the *Tropos4AS* models were not a simple refinement of the *Tropos* models, but built from scratch (see Appendix A, page 237). The questions can be answered mostly also by using the *Tropos* model only. They were discussed and corrected after a pilot study performed with subjects which did not participate to the experiment.

Moreover, the data analysis of the answers to the comprehension questions involved some subjectivity. It was performed by an expert (the author), by mapping the answers with a list of expected answers. Additional concepts, outside the requested ones, were not counted as such, in the evaluation. This analysis should possibly be performed by further persons and discussed, to remedy this possible threat to validity. To encourage the repetition of the experiment with other subjects, we make available the replication package online at `http://selab.fbk.eu/morandini/T4AScomprehensionexperiment_package.zip`.

8.8 Related Work

In a recent work, Y. Brun [Brun, 2010] analyses the current state of publications dealing with self-adaptive systems. He ascribes their low impact in current premier software engineering conferences and journals to the lack of a proper evaluation of the proposed ideas, techniques and methods and the difficulty of a head-to-head comparison with traditional techniques.

The evaluation we performed can be seen as a first step to fill this gap, not only for evaluating *Tropos4AS*, but also as a pilot study to understand how experts and novices make use of the *Tropos* modelling language. The work in [Hadar et al., 2010] empirically evaluates the goal-oriented approach *Tropos* against the scenario-based UML *Use Cases*, with a positive outcome for the goal modelling language. However, to achieve comparable, objective results, Hadar et al. need to explicitly limit the experiment to the comprehension of few constructs and small modification tasks. On the contrary, a modelling task could capture how people actually and intuitively use the language to display requirements graphically, and the modelling effort needed.

A similar comprehension study by Ricca et al [Ricca et al., 2010, Ricca et al., 2006] evaluates a type of diagrams (here, UML class diagrams) with one of its extensions (as an important analogy to our study). The study bases on example diagrams which were semi-automatically reverse-engineered from code. However, for our aim of extending an existing modelling language, an important, critical point to get expressive results would also be to evaluate the effort of modelling and the usage of the available modelling concepts.

Shehory and Sturm [Shehory and Sturm, 2001] propose a set of criteria to evaluate the quality of a methodology. Important criteria identified are, among others, preciseness, expressiveness, modularity, executability and refinability, from a software engineering viewpoint, and, from an agent orientation viewpoint, autonomy, adaptability and complexity. Such an evaluation would be interesting also for our new framework, but it is not so suitable for having a comparison of a modelling language with its extension.

Various works deal with the evaluation of goal-oriented modelling frameworks, first of all KAOS and *i**. Estrada et al. [Estrada et al., 2006] present the results of an in-depth empirical evaluation of *i**, the basis of *Tropos*, on industrial case studies. The evaluation (which was not conducted as an experiment with subjects) was carried out over a long time period with three teams of professional requirements engineers, evaluating the coverage of different aspects of the modelling language in-the-field. Reusability and scalability were recognized to be missing aspects, which remain still present in both *Tropos* and *Tropos4AS*, but *Tropos4AS* adds some of the expected granularity in the models.

Matulevicius and Heymans [Matulevicius and Heymans, 2007] compare the two prominent goal modelling languages *i** and *KAOS* by an experiment consisting of three steps: interviewing, creating goal models and evaluating models and languages. In groups, students interviewed persons that played the stakeholders and then delivered the requirements model created with the assigned language, within two weeks. The students then individually evaluated the used language and the model of a competing group, with questionnaires. Similarly to ours, the experiment was evaluated with Wilcoxon tests, following [Wohlin et al., 2000]. They conclude that the models created with *i** are evaluated better than KAOS models, whereas as language, KAOS is evaluated slightly better than *i**. The experiment includes a realistic, detailed approach to modelling and well-structured questionnaires. However, the results are based on only two models per methodology and thus the statistical tests are questionably applied to the single set of answers of each subject, while our analysis is performed (as usually done) on the subject answers for each question. It is undoubtedly a drawback that our subjects would not have been available for longer experiments. However, with our experimental design we were able to get modelling feedback from a variety of participants, from programmers to RE experts, and could do evaluations and statistical tests on them.

Our framework also includes an implementation in an agent programming language. Thus, as mentioned in the introduction of this section, an empirical study on the agent code produced with the framework, was also taken into consideration, but abandoned because of the complexity of the resulting code and the lack of subjects that are experienced in agent programming. However, there is a lack on empirical studies for the use of such languages in general, to justify the claims that motivate the introduced high-level constructs, which are also used in our framework. A first empirical study on the practical use of an agent programming language was performed by Riemsdijk and Hindriks [van Riemsdijk and Hindriks, 2009], trying to motivate the claim that high-level notions such as goals and beliefs provide appropriate abstractions to develop autonomous software. The study is performed with three programmers implementing small examples, which were then analysed quantitatively and qualitatively. Moreover, six subjects commented on readability of the code. Various observations on the use of the programming constructs by expert and non-expert programmers were made, although the small numbers did not permit any statistical analysis.

8.9 Final Considerations

We performed an empirical study, with the aim of evaluating the effectiveness of the *Tropos4AS* modelling language in comparison to *Tropos* modelling. The study included a first experiment based on a modelling task on self-adaptive systems, and a second experiment based on a comprehension task on models of the same systems, both performed by subjects from a research centre. The experimental settings and the short training on the modelling languages have shown to be adequate, giving enough expertise to solve the tasks properly.

We can conclude, accepting the alternative hypotheses with statistical evidence: Regarding research question *RQ*1, the effort required to apply *Tropos4AS* is higher than the effort required to apply *Tropos*. However, this additional effort is not perceived as such by the users, and they did not face particular difficulties (as evaluated with the questionnaire). The additional time was spent mainly for studying the *Tropos4AS* specifications, as the recorded times show. Regarding *RQ*2, *Tropos4AS* allows the users to produce models more effective and correct than *Tropos* for representing requirements of an adaptive system.

Finally, with a second study we were able to give a statistically significant answer to the research question *RQ*3 (Section 8.6.1): *Tropos4AS* models are more effective for the comprehension of system requirements than *Tropos* models. Moreover, by a subjective analysis of the models, we can say that *Tropos* and the *Tropos4AS* extensions were used as expected, by nearly all participants, including the ones which previously had no knowledge of *Tropos*. Although, a better training for both languages could improve the quality of the models.

The results of this empirical evaluation meet our expectations and strengthen them. In particular, the use of *Tropos* and the *Tropos4AS* extensions by novice users was better than expected and their comments show that the additional constructs bring few more complexity, but facilitate to express a set of requirements with applications having to adapt to the environment such as the examples. This was confirmed, from a different viewpoint, by the comprehension experiment.

However, we have to remark that the results of the study are limited to the use of the modelling languages and are thus not generalisable to the modelling process. Moreover, the small example systems used as objects of the study had a specific focus on self-adaptivity and thus the findings cannot be generalised to every kind of model, e.g. if the focus is on actor dependency diagrams. Also, the experiments were performed on small examples and thus not considering scalability issues that are expected in larger applications.

Chapter 9

Conclusion

9.1 Summary and Contributions

Among the challenges of developing self-adaptive systems, this work identified as key one that of capturing the knowledge necessary for a system to self-adapt and of bringing this knowledge to run-time. We introduced the *Tropos4AS* framework, which is described in detail in the Chapters 3 and 4 of this thesis. *Tropos4AS* extends *Tropos* goal modelling along different lines, enabling to model: *i*) the environment, capturing the influence of artifacts in the surroundings of an actor in the system to the actor's goals and their achievement process; *ii*) the modelling of goal types and inter-goal relationships, to detail goal achievement and alternatives selection dynamics; and *iii*) the modelling of possible failures, errors and proper recovery activities, to make the system more robust by eliciting missing functionalities, to separate the exceptional from the nominal behaviour of the system, and to create an interface for domain-specific diagnosis techniques.

The combination of (*i* – *iii*) contributes to our first research objective (given in Section 1.2), giving to the designer a conceptual tool to capture, with intuitive, comprehensible concepts, specific knowledge and decision criteria necessary to increase self-adaptivity in a system at run-time. This is a first, relevant step towards making the system aware of the objectives it has been designed for, and able to carry out decisions on the proper behaviour to exhibit at run-time. Thereby, we consider adaptivity which originates from variability and non-determinism in the requirements, while we do not consider run-time changes in the high-level requirements.

With environment and failure modelling, the framework anticipates some of the issues tackled also in recent works specific to self-adaptivity, for example

[Qureshi and Perini, 2009] and [Baresi and Pasquale, 2010]. However, both of these works lack of a direct mapping of these concepts to run-time.

The *Tropos4AS* modelling extensions were then integrated to a process which spans the development phases until the implementation, focussing on knowledge-level artefacts (Chapter 4). This process includes a mapping from the design artefacts to a BDI agent architecture (adopting the Jadex agent definition language), which preserves the representation of the requirements in form of a goal model at run-time, taking advantage of the features provided by Jadex, and extending it for reflecting the intended behaviour captured in the goal model.

Contributing to our second research objective (Section 1.2), by representing its requirements model at run-time, we give to an agent the awareness of 'why' to exhibit some behaviour, a core requisite for self-adaptivity. Moreover, a causal dependency between high-level requirements and executed functionalities is maintained. Supporting tools for conceptual modelling and for automated code generation for the Jadex agent platform, were presented. The implementation provides a goal model directed execution and basic mechanisms for goal model reasoning, monitoring and adaptation, for executing agent prototypes, which can be used to validate the model by observing its run-time behaviour. The obtained goal model can be navigated to improve the decision making, to allow the system for monitoring its goals and possibly also for modifying them. Moreover, the representation of the goal model at run-time eases traceability of artefacts and decisions back to the design and the requirements.

The *Tropos4AS* framework offers conceptual models for capturing the information necessary to carry out an adaptation, but does not focus on delivering specific run-time algorithms and mechanisms to perform the monitoring and decision making and to enact the adaptation. Thus, the proposed mapping to a BDI agent architecture should be considered as a basic architecture which can be customized with more sophisticated reasoning, monitoring and adaptation mechanisms, which take advantage of the mapped knowledge. Concretely, alternatives selection could be improved by implementing sophisticated goal reasoning mechanisms, learning mechanisms or including risk analysis, whereas the selection of recovery activities could be improved by implementing diagnosis mechanisms. Also, the run-time goal deliberation mechanisms can be extended.

Abstract operational semantics to define the attitude of an agent towards the satisfaction of various types of goals inside a goal model, were defined in Chapter 5, to set the basis for a formal definition of both the *Tropos4AS* goal models and for an implementation (i.e. agent platforms) supporting goal models at run-time. This work contributes to the state of the art by extending the unifying semantics for basic goal types in agent platforms defined in [van Riemsdijk et al., 2008], and differentiates from works such as [Thangarajah et al., 2010] by including goal decomposition.

The main contribution of Chapter 6 consists in the combination of a "top-down" goal-oriented modelling approach (*Tropos4AS*) with a "bottom-up" interaction-based approach (ADELFE), to achieve adaptation by agent self-organisation. From the viewpoint of ADELFE, this approach reduces the gap between global goals and the single agent's behaviour, while for *Tropos* and *Tropos4AS*, the main novelty lies in a flexible definition of agent interaction on an instance level. This is achieved by handling interaction failures, to optimize the dependencies between agents. Further research is required to assess the extent of the remaining gap between low-level agent goals and the behaviour arising from changing peer agents for interaction.

With an empirical study, in Chapter 8 we quantitatively evaluated the usability of the modelling concepts newly introduced with *Tropos4AS*, in comparison to standard *Tropos*, for modelling systems with self-adaptivity properties. Few controlled experiments with subjects on goal modelling, and on *Tropos* in particular, are available in literature. To our knowledge, this is the first comparative experiment which analyses both modelling and model comprehension, and obtains statistically significant data. From the feedback of the subjects we can conclude that *Tropos4AS* models are more comprehensible and more effective in capturing requirements, while the modelling effort does not seem to have been significantly higher.

9.2 Conclusion and Future Work

Addressing the research problems identified as thesis objectives, the tool-supported framework *Tropos4AS* presented in this thesis makes an important step towards an effective development of self-adaptive software systems.

First, the goal-oriented engineering approach used by *Tropos4AS* extends *Tropos* to effectively deal with requirements for self-adaptive systems, by capturing, on a knowledge level, the perceived environment and its effects on the goal achievement process, and by supporting the designer in the analysis of possible failures, errors causing them, and recovery activities.

Second, the development process preserves the concepts of *agent* and *goal model* through all development phases until run-time, lowering conceptual gaps and simplifying traceability of decisions and changes from requirements to code and vice-versa. Our approach captures adaptivity, modelling alternative behaviours and selection criteria at design time, which can then be exploited at run-time. Techniques that further optimize adaptation at run-time, such as reasoning on goal models and learning, are not covered in this thesis, but would complement our approach and can be added when needed.

The work provides an early investigation on aspects that were pointed out later on in recent research agendas for engineering of self-adaptive systems [Di Nitto et al., 2008, Cheng et al., 2009a, Sawyer et al., 2010]. Our results provide a first approach for modelling and for bringing requirements to run-time. Although specific to agent-based frameworks, they may be generalized to other implementation platforms.

Modelling tools and a tool-supported mapping from *Tropos4AS* models to a BDI agent architecture realise a direct transition from the requirements models to code for the *Jadex* agent platform, to deliver an explicit representation of requirements at run-time. For a generalization to non-BDI systems, the implementation of an appropriate middleware layer with a run-time reasoning loop for monitoring and goal evaluation would be necessary.

The modelling process and the mapping were applied to various examples, and a controlled experiment with subjects, carried out to collect empirical evidence about the effectiveness of *Tropos4AS*, gave positive, statistically relevant results, both for the modelling of system specifications and for model comprehension. The ***t2x*** code generation tool has been used in university courses for an introduction to BDI agents programming, by students for their course projects, and in master theses.

Several open issues are left, which are worth to be investigated in future. We mention principal research directions. To further consolidate *Tropos4AS*, the modelling process has to be validated on bigger case studies in dynamic domains. Here, also scalability issues have to be considered, even though the approach limits to the single agents in a system. Also, the complete goal model semantics should be defined by a formal mapping to a well-defined BDI language. If Jadex is used as a target language, its goal satisfaction semantics need to be precisely determined and captured e.g. with the operational semantics defined in this thesis. A formalisation obtained in this way could also be used to validate models and identify conflicts.

The automated mapping towards a BDI architecture has to be tested on bigger examples, in which a more complex behaviour is needed. A model-driven approach supporting round-trip engineering would be of high interest for keeping the design models synchronized with the implementation. It would also be interesting to investigate on failure diagnosis mechanisms and the discovery of proper recovery activities at run-time, by combining available capabilities or services or by collaboration between agents.

A further research direction concerns the possibility to give to the software the ability to enrich and modify its requirements model at run-time, to achieve a self-adaptive behaviour by learning from failures and collaboration. Our vision is to give automated feedback from run-time to developers for improving the correspondent design time models. An automated evaluation of modelled softgoal contributions in running prototypes, to give feedback for improving them at design time, could be a first step toward this aim.

The framework could be generalized by mappings to different agent platforms or to service-oriented architectures. Performance and scalability problems, arising mainly due to the use of the Jadex language, could be reduced by a mapping to a more lightweight platform. Finally, the empirical study, which gave promising results with graduate students and researchers, would have to be repeated with different subjects, available for longer time periods, and with an evaluation of the obtained models by different experts, to achieve more general results.

Bibliography

[Ali et al., 2010] Ali, R., Dalpiaz, F., and Giorgini, P. (2010). A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458.

[Antoniol et al., 2002] Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983.

[Asnar et al., 2006] Asnar, Y., Bryl, V., and Giorgini, P. (2006). Using risk analysis to evaluate design alternatives. In *AOSE*, pages 140–155.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33.

[Bacchelli et al., 2010] Bacchelli, A., D'Ambros, M., and Lanza, M. (2010). Extracting source code from e-mails. In *ICPC*, pages 24–33.

[Baresi and Pasquale, 2010] Baresi, L. and Pasquale, L. (2010). Live goals for adaptive service compositions. In *SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 114–123, New York, NY, USA. ACM.

[Bellifemine et al., 2007] Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE.* Wiley.

[Bernon et al., 2005] Bernon, C., Camps, V., Gleizes, M.-P., and Picard, G. (2005). Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. In Henderson-Sellers, B. and Giorgini, P., editors, *Agent-Oriented Methodologies*, pages 172–202. Idea Group, NY, USA.

[Bernon et al., 2002] Bernon, C., Gleizes, M., Peyruqueou, S., and Picard, G. (2002). ADELFE, a Methodology for Adaptive Multi-Agent Systems Engineering. In *Third International Workshop Engineering Societies in the Agents World (ESAW-2002).*

[Berry et al., 2005] Berry, D., Cheng, B., and Zhang, J. (2005). The four levels of requirements engineering for and in dynamic adaptive systems. In *Proceedings of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality, Porto, Portugal.*

[Bordini et al., 2005] Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2005). *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations.* Springer.

[Bordini and Hübner, 2005] Bordini, R. H. and Hübner, J. F. (2005). Bdi agent programming in agentspeak using *jason* (tutorial paper). In *CLIMA VI*, pages 143–164.

[Borgida et al., 2009] Borgida, A., Chaudhri, V. K., Giorgini, P., and Yu, E. S. K., editors (2009). *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, volume 5600 of *LNCS*. Springer.

[Bratman, 1987] Bratman, M. E. (1987). *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, MA.

[Braubach et al., 2004] Braubach, L., Pokahr, A., Moldt, D., and Lamersdorf, W. (2004). Goal representation for bdi agent systems. In *PROMAS*, pages 44–65.

[Bresciani et al., 2004a] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2004a). Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

[Bresciani et al., 2004b] Bresciani, P., Penserini, L., Busetta, P., and Kuflik, T. (2004b). Agent patterns for ambient intelligence. In *23th International Conference on Conceptual Modeling (ER 2004)*, LNCS 3288, pages 682 – 695, Shanghai, China. Springer-Verlag.

[Brun, 2010] Brun, Y. (2010). Improving impact of self-adaptation and self-management research through evaluation methodology. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS10)*, pages 1–9.

[Capera et al., 2003] Capera, D., Georgé, J.-P., Gleizes, M.-P., and Glize, P. (2003). The AMAS Theory for Complex Problem Solving Based on Self-organizing Cooperative Agents . In *TAPOCS 2003 at WETICE 2003, Linz, Austria, 09/06/03-11/06/03*. IEEE CS.

[Cares et al., 2005] Cares, C., Franch, X., and Mayol, E. (2005). Extending tropos for a prolog implementation: A case study using the food collecting agent problem. In *CLIMA VI*, pages 396–405.

[Castro et al., 2002] Castro, J., Kolp, M., and Mylopoulos, J. (2002). Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*. Elsevier, Amsterdam, the Netherlands.

[Cheng et al., 2009a] Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors (2009a). *Software Engineering for Self-Adaptive Systems (outcome of a Dagstuhl Seminar)*, volume 5525 of *Lecture Notes in Computer Science*. Springer.

[Cheng et al., 2009b] Cheng, B. H. C., Sawyer, P., Bencomo, N., and Whittle, J. (2009b). A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *MODELS '09: 12th Int. Conference on Model Driven Engineering Languages and Systems*, pages 468–483. Springer-Verlag.

[Cheng et al., 2004] Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B. R., and Steenkiste, P. (2004). Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In *ICAC*, pages 276–277.

[Chopra et al., 2010] Chopra, A. K., Dalpiaz, F., Giorgini, P., and Mylopoulos, J. (2010). Reasoning about agents and protocols via goals and commitments. In *AAMAS*, pages 457–464.

[Cohen, 2004] Cohen, J. (2004). *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ.

[Dalgaard, 2008] Dalgaard, P. (2008). *Introductory Statistics with R (Statistics and Computing)*. Springer, 2nd edition.

[Dalpiaz et al., 2010] Dalpiaz, F., Chopra, A. K., Giorgini, P., and Mylopoulos, J. (2010). Adaptation in open systems: Giving interaction its rightful place. In *ER*, pages 31–45.

[Dalpiaz et al., 2009] Dalpiaz, F., Giorgini, P., and Mylopoulos, J. (2009). An architecture for requirements-driven self-reconfiguration. In *CAiSE*, pages 246–260.

[Dardenne et al., 1993] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-directed requirements acquisition. In *Selected Papers of the 6. Int. Workshop on Software Specification and Design*, pages 3–50, Amsterdam, The Netherlands. Elsevier.

[Darimont et al., 1997] Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1997). Grail/kaos: An environment for goal-driven requirements engineering. In *ICSE*, pages 612–613.

[Dastani, 2008] Dastani, M. (2008). 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

[Dastani et al., 2006] Dastani, M., van Riemsdijk, M. B., and Meyer, J.-J. C. (2006). Goal types in agent programming. In *ECAI*, pages 220–224.

[DeLoach, 2002] DeLoach, S. A. (2002). Modeling organizational rules in the multi-agent systems engineering methodology. In *Canadian Conference on AI*, pages 1–15.

[DeLoach and Miller, 2009] DeLoach, S. A. and Miller, M. (2009). A goal model for adaptive complex systems. In *International Conference on Knowledge-Intensive Multi-Agent Systems (KIMAS 2009)*, St. Louis, MO.

[DeLoach et al., 2009] DeLoach, S. A., Padgham, L., Perini, A., Susi, A., and Thangarajah, J. (2009). Using three aose toolkits to develop a sample design. *Int. J. Agent-Oriented Softw. Eng.*, 3(4):416–476.

[DeLoach et al., 2001] DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258.

[Di Nitto et al., 2008] Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M. P., and Pohl, K. (2008). A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341.

[Duff et al., 2006] Duff, S., Harland, J., and Thangarajah, J. (2006). On proactivity and maintenance goals. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1033–1040, New York, NY, USA. ACM.

[Estrada et al., 2006] Estrada, H., Rebollar, A. M., Pastor, O., and Mylopoulos, J. (2006). An empirical evaluation of the *i** framework in a model-based software generation environment. In *CAiSE*, pages 513–527.

[Feather et al., 1998] Feather, M., Fickas, S., van Lamsweerde, A., and Ponsard., C. (1998). Reconciling System Requirements and Runtime Behaviour. In *9th IEEE Int. Workshop on Software Specification and Design (IWSSD-98)*, Isobe, Japan.

[Fuxman et al., 2004] Fuxman, A., Liu, L., Mylopoulos, J., Roveri, M., and Traverso, P. (2004). Specifying and analyzing early requirements in tropos. *Requir. Eng.*, 9(2):132–150.

[Fuxman et al., 2001] Fuxman, A., Pistore, M., Mylopoulos, J., and Traverso, P. (2001). Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA).

[Ganek and Corbi, 2003] Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18.

[García-Ojeda et al., 2007] García-Ojeda, J. C., DeLoach, S. A., Robby, Oyenan, W. H., and Valenzuela, J. (2007). O-mase: A customizable approach to developing multiagent development processes. In *AOSE*, pages 1–15.

[Gardelli et al., 2008] Gardelli, L., Viroli, M., Casadei, M., and Omicini, A. (2008). Designing self-organising environments with agents and artefacts: a simulation-driven approach. *IJAOSE*, 2(2):171–195.

[Gershenson, 2005] Gershenson, C. (2005). A general methodology for designing self-organizing systems. *CoRR*, abs/nlin/0505009.

[Giorgini et al., 2005a] Giorgini, P., Massacci, F., Mylopoulos, J., and Zannone, N. (2005a). Modeling security requirements through ownership, permission and delegation. In *RE*, pages 167–176.

[Giorgini et al., 2004] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., and Sebastiani, R. (2004). Formal Reasoning Techniques for Goal Models. *Journal on Data Semantics.*

[Giorgini et al., 2005b] Giorgini, P., Mylopoulous, J., and Sebastiani, R. (2005b). Goal-Oriented Requirements Analysis and Reasoning in the Tropos Methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159–171.

[Goldsby et al., 2008] Goldsby, H. J., Sawyer, P., Bencomo, N., Hughes, D., and Cheng., B. H. C. (2008). Goal-based modeling of dynamically adaptive system requirements. In *ECBS 08, Belfast, Northern Ireland*.

[Hadar et al., 2010] Hadar, I., Kuflik, T., Perini, A., Reinhartz-Berger, I., Ricca, F., and Susi, A. (2010). An empirical study of requirements model understanding: Use Case vs. Tropos models. In *SAC*, pages 2324–2329.

[Henderson-Sellers and Giorgini, 2005] Henderson-Sellers, B. and Giorgini, P., editors (2005). *Agent-Oriented Methodologies*. Idea Group Inc.

[Hindriks et al., 1999] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1999). Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.

[Hindriks et al., 2000] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (2000). Agent programming with declarative goals. In *7th Int. Workshop on Agent Theories Architectures and Languages (ATAL 2000), Boston, MA, USA*, pages 228–243.

[Hindriks and van Riemsdijk, 2007] Hindriks, K. V. and van Riemsdijk, M. B. (2007). Satisfying maintenance goals. In *DALT*, pages 86–103.

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

[Jennings, 2000] Jennings, N. R. (2000). On agent-based software engineering. *Artif. Intell.*, 117(2):277–296.

[Jureta et al., 2010] Jureta, I. J., Borgida, A., Ernst, N. A., and Mylopoulos, J. (2010). Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *18th IEEE Int. Requirements Engineering Conference*, pages 115–124, Sydney, Australia.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41–50.

[Khallouf and Winikoff, 2009] Khallouf, J. and Winikoff, M. (2009). The goal-oriented design of agent systems: a refinement of prometheus and its evaluation. *IJAOSE*, 3(1):88–112.

[Kiessel et al., 2002] Kiessel, J., Beard, J., and Nielsen, P. (2002). Failure recovery: A software engineering methodology for robust agents. In *SELMAS 2002, Orlando, Florida, USA*.

[Kinny et al., 1996] Kinny, D., Georgeff, M. P., and Rao, A. S. (1996). A methodology and modelling technique for systems of bdi agents. In *MAAMAW*, pages 56–71.

[Kolp et al., 2001] Kolp, M., Giorgini, P., and Mylopoulos, J. (2001). A goal-based organizational perspective on multi-agents architectures. In *Proceedings of the Eighth International Workshop on Agent Theories, architectures, and languages (ATAL-2001)*.

[Krishna et al., 2006] Krishna, A., Guan, Y., and Ghose, A. K. (2006). Co-evolution of i* models and 3apl agents. In *Sixth International Conference on Quality Software (QSIC 2006)*, pages 117–124.

[Laddaga, 2006] Laddaga, R. (2006). Self adaptive software problems and projects. In *Proceedings of the 2nd International IEEE Workshop on Software Evolvability (SE'06)*, pages 3–10, Washington, DC, USA. IEEE Computer Society.

[Lapouchnian et al., 2006] Lapouchnian, A., Yu, Y., Liaskos, S., and Mylopoulos, J. (2006). Requirements-driven design of autonomic application software. In *CASCON*, pages 80–94.

[Liaskos et al., 2006] Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., and Mylopoulos, J. (2006). On goal-based variability acquisition and analysis. In *14th IEEE International Conference on Requirements Engineering*, Minneapolis.

[Liaskos and Mylopoulos, 2010] Liaskos, S. and Mylopoulos, J. (2010). On temporally annotating goal models. In *4th International i* Workshop (i* 2010)*.

[Maes, 1994] Maes, P. (1994). Modeling adaptive autonomous agents. *Artificial Life*, 1(1-2):135–162.

[Matulevicius and Heymans, 2007] Matulevicius, R. and Heymans, P. (2007). Comparing goal modelling languages: An experiment. In *REFSQ*, pages 18–32.

[Mermet and Simon, 2009] Mermet, B. and Simon, G. (2009). Gdt4mas: an extension of the gdt model to specify and to verify multiagent systems. In *AAMAS (1)*, pages 505–512.

[Molesini et al., 2005] Molesini, A., Omicini, A., Denti, E., and Ricci, A. (2005). Soda: A roadmap to artefacts. In *ESAW*, pages 49–62.

[Morandini, 2006] Morandini, M. (2006). Knowledge level engineering of BDI agents. Master's thesis, DIT, Università di Trento, Italy. Available at `http://disi.unitn.it/~morandini/resources/ThesisMirkoMorandini.pdf`.

[Morandini et al., 2009a] Morandini, M., Migeon, F., Gleizes, M.-P., Maurel, C., Penserini, L., and Perini, A. (2009a). A Goal-Oriented Approach for Modelling Self-Organising MAS. In *Proceedings of the 10th International Workshop on Engineering Societies in the Agents' World (ESAW 2009)*, volume 5881 of *LNCS*. Springer.

[Morandini et al., 2008a] Morandini, M., Nguyen, D. C., Perini, A., Siena, A., and Susi, A. (2008a). Tool-supported development with tropos: The conference management system case study. In Luck, M. and Padgham, L., editors, *Agent Oriented Software Engineering VIII*, volume 4951 of *LNCS*, pages 182–196. Springer. 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 2007.

[Morandini et al., 2008b] Morandini, M., Penserini, L., and Perini, A. (2008b). Automated mapping from goal models to self-adaptive systems. In *Demo session at the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 485–486.

[Morandini et al., 2008c] Morandini, M., Penserini, L., and Perini, A. (2008c). Modelling self-adaptivity: A goal-oriented approach. In *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'08)*, pages 469–470. IEEE.

[Morandini et al., 2008d] Morandini, M., Penserini, L., and Perini, A. (2008d). Towards goal-oriented development of self-adaptive systems. In *SEAMS '08: Workshop on Software engineering for adaptive and self-managing systems, colocated with ICSE 2008*, pages 9–16, New York, NY, USA. ACM.

[Morandini et al., 2009b] Morandini, M., Penserini, L., and Perini, A. (2009b). Operational Semantics of Goal Models in Adaptive Agents. In *8th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'09)*. IFAAMAS.

[Morandini et al., 2010] Morandini, M., Penserini, L., and Perini, A. (2010). Goal-oriented development of self-adaptive systems. Technical report, Fondazione Bruno

Kessler, Trento, Italy. Submitted to the Journal of Information and Software Technology, under review.

[Morandini et al., 2008e] Morandini, M., Penserini, L., Perini, A., and Susi, A. (2008e). Refining goal models by evaluating system behaviour. In Luck, M. and Padgham, L., editors, *Agent Oriented Software Engineering VIII*, volume 4951 of *LNCS*, pages 44–57. Springer. 8th Int. Workshop, AOSE 2007, Honolulu, HI, USA, May 2007.

[Müller et al., 2008] Müller, H., Pezzè, M., and Shaw, M. (2008). Visibility of control in adaptive systems. In *Proceedings of the 2nd Int. Workshop on Ultra-large-scale software-intensive systems (ULSSIS'08)*, pages 23–26, New York, NY, USA. ACM.

[Nakagawa et al., 2008] Nakagawa, H., Ohsuga, A., and Honiden, S. (2008). Constructing self-adaptive systems using a kaos model. In *Proc. of the 2nd IEEE Self-Adaptive and Self-Organizing Systems Workshops (SASOW '08)*, pages 132–137. IEEE.

[Nakagawa et al., 2010] Nakagawa, H., Ohsuga, A., and Honiden, S. (2010). Cooperative behaviors description for self-* systems implementation. In *PAAMS*, pages 69–74.

[Nguyen et al., 2009] Nguyen, C. D., Miles, S., Perini, A., Tonella, P., Harman, M., and Luck, M. (2009). Evolutionary testing of autonomous software agents. In *8th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 521–528. IFAAMAS.

[Nguyen et al., 2008] Nguyen, C. D., Perini, A., and Tonella, P. (2008). ecat: a tool for automating test cases generation and execution in testing multi-agent systems (demo paper). In *7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 1669–1670. IFAAMS.

[Nguyen et al., 2010] Nguyen, C. D., Perini, A., and Tonella, P. (2010). Goal-oriented testing for MASs. *Int. J. Agent-Oriented Software Engineering*, 4(1):79–109.

[O'Brien and Nicol, 1998] O'Brien, P. and Nicol, R. (1998). FIPA - Towards a standard for software agents. *BT Technology Journal*, 16(3):51–59.

[Omicini et al., 2006] Omicini, A., Ricci, A., and Viroli, M. (2006). *Agens Faber*: Toward a theory of artefacts for MAS. *Electr. Notes Theor. Comput. Sci.*, 150(3):21–36.

[Oyenan and DeLoach, 2010] Oyenan, W. H. and DeLoach, S. A. (2010). Towards a systematic approach for designing autonomic systems. *Web Intelligence and Agent Systems*, 8(1):79–97.

[Padgham and Winikoff, 2002] Padgham, L. and Winikoff, M. (2002). Prometheus: a methodology for developing intelligent agents. In *AAMAS*, pages 37–38.

[Padgham et al., 2008] Padgham, L., Winikoff, M., DeLoach, S. A., and Cossentino, M. (2008). A unified graphical notation for aose. In *AOSE*, pages 116–130.

[Parhami, 1997] Parhami, B. (1997). Defect, fault, error,..., or failure? *IEEE Transactions on Reliability*, 46(4):450–451.

[Pavón et al., 2008] Pavón, J., Sansores, C., and Gómez-Sanz, J. J. (2008). Modelling and simulation of social systems with ingenias. *IJAOSE*, 2(2):196–221.

[Penserini et al., 2006a] Penserini, L., Kolp, M., and Spalazzi, L. (2006a). Social-oriented engineering of intelligent software. *WIAS: Web Intelligence and Agent Systems: An International Journal.*, IOS Press.

[Penserini et al., 2010] Penserini, L., Kuflik, T., Busetta, P., and Bresciani, P. (2010). Agent-based organizational structures for ambient intelligence scenarios. *JAISE*, 2(4):409–433.

[Penserini et al., 2007a] Penserini, L., Perini, A., Susi, A., Morandini, M., and Mylopoulos, J. (2007a). A Design Framework for Generating BDI-Agents from Goal Models. In *6th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'07), Honolulu, Hawaii*, pages 610–612.

[Penserini et al., 2006b] Penserini, L., Perini, A., Susi, A., and Mylopoulos, J. (2006b). From capability specifications to code for multi-agent software. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE-06)*, pages 253 – 256, Tokyo, Japan. IEEE press.

[Penserini et al., 2006c] Penserini, L., Perini, A., Susi, A., and Mylopoulos, J. (2006c). From Stakeholder Intentions to Software Agent Implementations. In *Proceedings of the 18th Conference On Advanced Information Systems Engineering (CAiSE'06)*, volume 4001 of *LNCS*, pages 465–479, Luxemburg. Springer-Verlag.

[Penserini et al., 2007b] Penserini, L., Perini, A., Susi, A., and Mylopoulos, J. (2007b). High variability design for software agents: Extending tropos. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4).

[Perini and Susi, 2004] Perini, A. and Susi, A. (2004). Developing Tools for Agent-Oriented Visual Modeling. In Lindemann, G., Denzinger, J., Timm, I., and Unland, R., editors, *Multiagent System Technologies, Proc. of the Second German Conference, MATES 2004*, number 3187 in LNAI, pages 169–182. Springer-Verlag.

[Picard et al., 2005] Picard, G., Bernon, C., and Gleizes, M.-P. (2005). Etto: Emergent timetabling organization. In *International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS), Budapest, Hungary*, pages 440–449. Springer.

[Pokahr et al., 2003] Pokahr, A., Braubach, L., and Lamersdorf, W. (2003). Jadex: Implementing a bdi-infrastructure for jade agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85.

[Pokahr et al., 2005] Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A bdi reasoning engine. In R. Bordini, M. Dastani, J. D. and Seghrouchni, A. F., editors, *Multi-Agent Programming*, pages 149–174. Springer, USA. Book chapter.

[Qureshi and Perini, 2010] Qureshi, N. and Perini, A. (2010). Continuous adaptive requirements engineering: An architecture for self-adaptive service-based applications. In *RE@RunTime Workshop at RE'10*, pages 17–24, Sydney, Australia.

[Qureshi et al., 2010a] Qureshi, N., Perini, A., Ernst, N., and Mylopoulos, J. (2010a). Towards a continuous requirements engineering framework for self-adaptive systems. In *RE@RunTime Workshop at RE'10*, pages 9–16, Sydney, Australia.

[Qureshi et al., 2010b] Qureshi, N. A., Morandini, M., Nguyen, C. D., and Perini, A. (2010b). A tool-supported development process for adaptive systems. Technical report, SE unit, Fondazione Bruno Kessler, Trento, Italy. Available at `http://disi.unitn.it/~morandini/resources/TRQureshiMorandini10.pdf`.

[Qureshi and Perini, 2009] Qureshi, N. A. and Perini, A. (2009). Engineering adaptive requirements. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*, pages 126–131, Vancouver, BC.

[Rao and Georgeff, 1995] Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: From theory to practice. In *ICMAS*, pages 312–319.

[Ricca et al., 2006] Ricca, F., Penta, M. D., Torchiano, M., Tonella, P., and Ceccato, M. (2006). An empirical study on the usefulness of conallen's stereotypes inweb application comprehension. In *WSE*, pages 58–68.

[Ricca et al., 2010] Ricca, F., Penta, M. D., Torchiano, M., Tonella, P., and Ceccato, M. (2010). How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. *IEEE Transactions on Software Engineering*, 36(1):96–118.

[Rougemaille, 2008] Rougemaille, S. (2008). *Model Driven Engineering for Adaptive Multi-Agent Systems*. PhD thesis, Université Paul Sabatier, Toulouse, France.

[Salehie, 2009] Salehie, M. (2009). *A Quality-Driven Approach to Enable Decision-Making in Self-Adaptive Software*. PhD thesis, University of Waterloo, Ontario, Canada.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2).

[Sardina and Padgham, 2010] Sardina, S. and Padgham, L. (2010). A BDI agent progarmming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*.

[Sawyer et al., 2007] Sawyer, P., Bencomo, N., Hughes, D., Grace, P., Goldsby, H. J., and Cheng, B. H. C. (2007). Visualizing the analysis of dynamically adaptive systems using i* and dsls. In *2nd International Workshop on Requirements Engineering Visualization, New Delhi, India*.

[Sawyer et al., 2010] Sawyer, P., Bencomo, N., Whittle, J., Letier, E., and Finkelstein, A. (2010). Requirements-aware systems: A research agenda for re for self-adaptive systems. *IEEE Int. Conference on Requirements Engineering (RE2010)*, 0:95–103.

[Shehory and Sturm, 2001] Shehory, O. and Sturm, A. (2001). Evaluation of modeling techniques for agent-based systems. In *Agents*, pages 624–631.

[Shoham, 1993] Shoham, Y. (1993). Agent-Oriented Programming. *Artificial Intelligence*, 60:51 – 92.

[Simon et al., 2005] Simon, G., Mermet, B., and Fournier, D. (2005). Goal decomposition tree: An agent model to generate a validated agent behaviour. In *DALT*, pages 124–140.

[Souza et al., 2010] Souza, V., Lapouchnian, A., Robinson, W., and Mylopoulos, J. (2010). Awareness requirements for adaptive systems. Technical Report DISI-10-049, DISI, Università di Trento, Italy.

[Susi et al., 2005] Susi, A., Perini, A., Giorgini, P., and Mylopoulos, J. (2005). The Tropos Metamodel and its Use. *Informatica (Slovenia)*, 29(4):401–408.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

[Thangarajah et al., 2010] Thangarajah, J., Harland, J., Morley, D. N., and Yorke-Smith, N. (2010). On the life-cycle of bdi agent goals. In *ECAI*, pages 1031–1032.

[Tomasi, 2009] Tomasi, B. (2009). Improving the design of software agents based on the evaluation of run-time preferences. Master's thesis, DISI, Università di Trento, Italy. Advisors: Angelo Susi and Mirko Morandini. Available at `http://disi.unitn.it/~morandini/resources/ThesisBarbaraTomasi.pdf`.

[van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *RE*, page 249.

[van Lamsweerde, 2004] van Lamsweerde, A. (2004). Elaborating security requirements by construction of intentional anti-models. In *ICSE '04*, pages 148–157, Washington, USA. IEEE.

[van Lamsweerde and Letier, 2000] van Lamsweerde, A. and Letier, E. (2000). Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10).

[van Riemsdijk et al., 2008] van Riemsdijk, B., Dastani, M., and Winikoff, M. (2008). Goals in agent systems: A unifying framework. In *Proceedings of the 7th Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 713–720.

[van Riemsdijk et al., 2005] van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. C. (2005). Subgoal semantics in agent programming. In *EPIA*, pages 548–559.

[van Riemsdijk and Hindriks, 2009] van Riemsdijk, M. B. and Hindriks, K. V. (2009). An empirical study of agent programs. In *PRIMA*, pages 200–215.

[Wang et al., 2007] Wang, Y., McIlraith, S. A., Yu, Y., and Mylopoulos, J. (2007). An automated approach to monitoring and diagnosing requirements. In *ASE*, pages 293–302.

[Weyns et al., 2007] Weyns, D., Omicini, A., and Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30.

[Whittle et al., 2009] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H. C., and Bruel, J.-M. (2009). Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *17th IEEE International Requirements Engineering Conference, RE '09*, pages 79–88, Washington, DC, USA. IEEE Computer Society.

[Winikoff, 2005] Winikoff, M. (2005). Jack intelligent agents: An industrial strength platform. In [Bordini et al., 2005], pages 175–193.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction.* Kluwer Academic Publishers, Norwell, MA, USA.

[Wolf and Holvoet, 2005] Wolf, T. D. and Holvoet, T. (2005). Towards a methodology for engineering self-organising emergent systems. In *SOAS*, pages 18–34.

[Wooldridge, 1997] Wooldridge, M. (1997). Agent-based software engineering. *IEEE Proceedings - Software*, 144(1):26–37.

[Yu, 1995] Yu, E. (1995). *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Department of Computer Science.

[Yu et al., 2008] Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., and do Prado Leite, J. C. S. (2008). From goals to high-variability software design. In *ISMIS*, pages 1–16.

[Zambonelli et al., 2003] Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. *ACM Transactions on software Engineering and Methodology*, 12(3):317–370.

[Zhu et al., 2008] Zhu, Q., Lin, L., Kienle, H. M., and Müller, H. A. (2008). Characterizing maintainability concerns in autonomic element design. In *ICSM*, pages 197–206.

Appendix A

Empirical Study: Experiment Material

The following pages contain the replication packages for the two experiments carried out for the empirical study described in Chapter 8. First , from page 226, we present the material used for the modelling experiment. It includes a description of the assignment task, short descriptions of the two treatments (i.e. the goal modelling languages for *Tropos* and *Tropos4AS*), the objects in form of a "system story" defining the requirements of the systems PMA (Patient Monitoring Agent) and WMM (Washing Machine Manager), draft pages for the modelling, and the pre-, mid- and post-questionnaires. Second, from page 235, the material for the comprehension experiment is shown: A short description of the experiment task to execute, together with the textual requirements specifications for the systems (PMA and WMM); the "gold standard" models for these two systems, both for *Tropos* and for *Tropos4AS* (including conditions and failures), and the questionnaires (for PMA and WMM) containing the comprehension questions together with a short post-questionnaire. The anonymised raw data of the experiment results will be made available online at `http://selab.fbk.eu/morandini/T4ASmodellingexperiment_rawdata.zip` and `http://selab.fbk.eu/morandini/T4AScomprehensionexperiment_rawdata.zip`.

Empirical study: Tropos – Tropos4AS

Guide to the Experiment

You will be given the following sheets:

- A pre-questionnaire
- The description of example A and the corresponding questionnaire
- The description of example B and the corresponding questionnaire
- A final post-questionnaire

Your tasks:

1. Fill in the pre-questionnaire (page 2)
2. Write the starting time (on page 5A)
3. Carefully read the **whole** system story of the first example (page 4A)
4. Imagine to be an analyst and **model the requirements** of the first example with the first methodology assigned to you, as detailed as possible.
   - **Follow step by step the system story** (page 4A).
   - Use page 5A (and 6A, if you need) to draw the model
   - Cross-check the model by answering to the **control questions** provided (page 5A) and refine it if needed.
   - Please try to remember the relative **time** used for the different modelling activities you perform.
5. Write the end time (on page 5A)
6. Answer to the questionnaire (page 7A+8A)
7. **Repeat steps 2 to 6 for the second example (pages 3B-8B) with the second methodology assigned to you.**
8. Fill in the short post-questionnaire (page 9)

Thank you!

Assigned to you:

A) Example PMA	with methodology Tropos.
B) Example WMM	with methodology Tropos4AS.

Pre-Questionnaire

- Name & Surname: ____
- Position: ____________________________

Are you working on research in requirements engineering? ❑ Yes ❑ No

How much experience do you have in practice with requirements analysis?

❑ Few

❑ I worked as software analyst in research projects

❑ I worked as software analyst in industry

How much experience do you have with TROPOS modelling?

❑ None

❑ I modelled some small examples

❑ I am experienced in the use of TROPOS

How much experience do you have in general with the development of agent-oriented software?

❑ I never used any agent programming language

❑ I developed small examples. I know what a “BDI architecture” is.

❑ I developed agent-oriented systems with BDI architecture.

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

• I understood the basic notions of TROPOS modelling	❑ **1** ❑ **2** ❑ **3** ❑ **4** ❑ **5**
• The visual notation used in TROPOS is clear	❑ **1** ❑ **2** ❑ **3** ❑ **4** ❑ **5**
• The visual notation used in TROPOS4AS is clear	❑ **1** ❑ **2** ❑ **3** ❑ **4** ❑ **5**

Thank you for your collaboration!

The data of the questionnaires in this experiments will be used only for research purposes and will be divulged only in aggregated form.

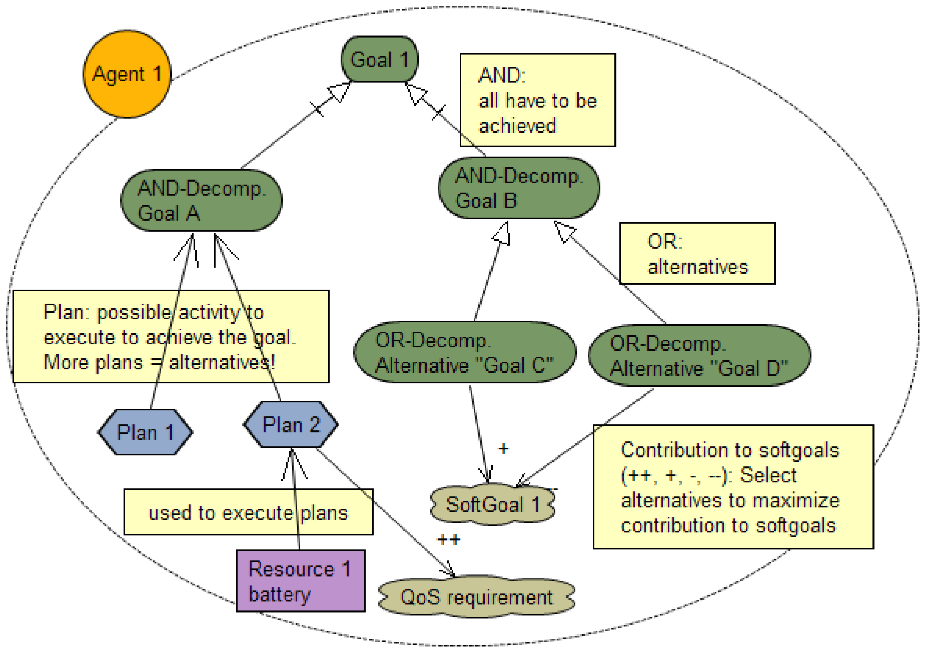
Date …………………………

Signature ____________________________

3A

Tropos

The main concepts of Tropos that can/should be used in the modelling experiment:


Agent 1
Goal 1
AND: all have to be achieved
AND-Decomp. Goal A
AND-Decomp. Goal B
OR: alternatives
Plan: possible activity to execute to achieve the goal. More plans = alternatives!
OR-Decomp. Alternative "Goal C"
OR-Decomp. Alternative "Goal D"
Plan 1
Plan 2
+
Contribution to softgoals (++, +, -, --): Select alternatives to maximize contribution to softgoals
used to execute plans
SoftGoal 1
++
Resource 1 battery
QoS requirement

Tropos models in a nutshell

Call goals and plans with names that describe their “content”!

- **Goals** can be decomposed to sub-**goals**, in AND or in OR.
- **Plans (“activities”)** can be decomposed to sub-**plans**, in AND or in OR.
- Plans are the means to achieve a goal.
  Plans in relation to a goal **are alternatives**!
- **Goals** and **Plans** can **contribute** to satisfice softgoals.
- Contribution can be positive to negative (++, +, -, --).
- Resources are used by plans.

- Semantics: By analysis of softgoal contribution, alternative goals and plans can be selected. If at run-time some goal or plan fails, alternatives can be tried!

Tropos goal modelling process

- Decompose goals, starting from the root goal, to more detailed, more concrete goals...
- ...until you are able to define plans that can achieve the goals.
- Define the Resources used, to execute the plans.
- Try to model contributions to the satisficement of softgoals.

4A
Example: Patient Monitoring Agent

System story

Summary: You have to develop an “intelligent” patient monitoring agent (PMA) software. The main aim of this software is to ensure that a patient, in a “smart home” environment, follows the medical instructions on eating meals and taking medicine. The system should **reduce the need for human assistance**, but **not bring annoyances** to the patient’s life.

The system can access to reliable **sensors** that are able to measure if the **patient takes the medicine** and **how often he eats**. The flat is set up with **loudspeakers** in each room. It is also connected to the **phone line** for the system to request assistance from the care-assistant.

Requirements: The PMA has the goal to ensure that the patient follows the medical instructions. This goal can be detailed to two sub-goals: eating meals regularly, and taking the medicine on schedule. The first sub-goal is achieved if the patient had **at least two meals a day**. The second one is achieved if the the medicine was taken in the evening, after 6PM, but fails if the medicine was not taken until 8PM. Moreover, the medicine has to be taken **after the last meal**.

The goal of eating meals regularly can be further detailed into eating breakfast (at 8AM), eating lunch (at noon) and eating dinner (at 6PM). How can each of these goals be achieved? For one hour, the patient should just be monitored to see if he eats by himself. In this way the system does not bring annoyances to accurate patients. Only after one hour the PMA should change its plan and remember the patient (through the loudspeakers), repeatedly. Obviously, this goal contributes negatively to the softgoal of not bringing annoyances, but still it contributes positively to reduce the need for human assistance.

If the patient does not eat the meal also after one hour of such requests, the respective plans will fail. If after dinner time the patient did still not have **at least two meals** (remember that this was the achievement condition of the goal of eating meals regularly), assistance from the case-assistant should be requested.

The system should remember the patient for taking the medicine. If the medicine was not taken until 8PM, the care-assistant should be called for assistance.

[Suggestion: first try to complete the model until here]

Now, suppose that the patient didn't eat for dinner before 8PM. Thus the subgoal of taking the medicine on schedule may fail (as defined above, he should not eat **after** taking the medicine). But the error could be prevented: With the precondition that the patient already took breakfast and lunch (and thus he already had two meals), he would be requested to take the medicine immediately, skipping the dinner. Try to model this and imagine other possible errors and activities to be done to prevent this failure!

Modelling Task

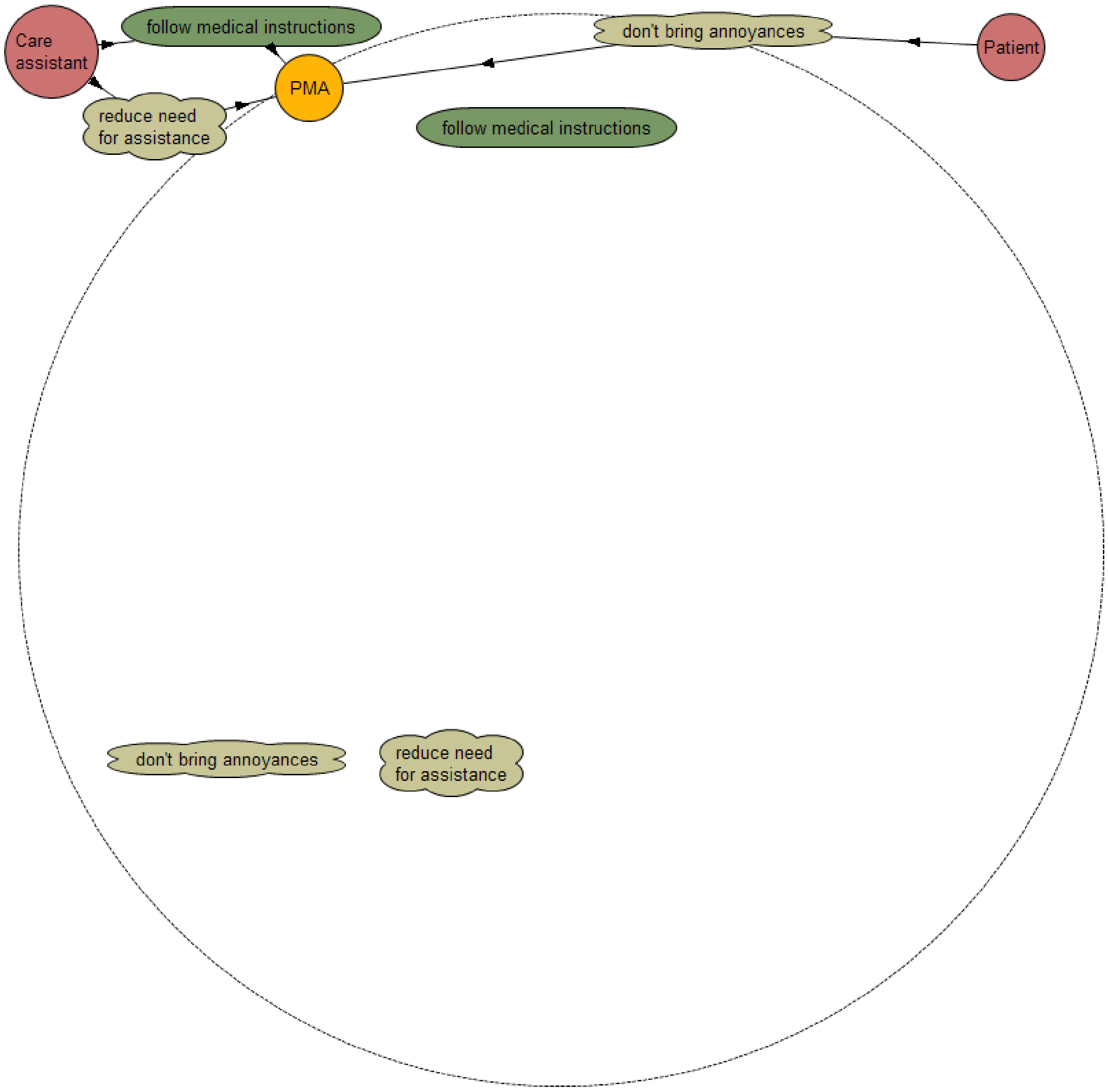
Try to model the requirements for the patient monitoring agent as detailed as possible, following **Tropos**, starting from the goal model of the actor “PMA” provided to you, which already contains the main goal and two softgoals. Draw the models by hand, following the modelling guidelines provided. Feel free to ask us questions on modelling constraints.

Model the example step by step following the system story. When modelling, try to freely interpret the requirements. There is no “right” or “expected” solution your models have to conform to. Remember to model contributions to the softgoals!

5A

Patient Monitoring Agent **Tropos** ***model***

Start time ________ ***End time*** ________


Care assistant
follow medical instructions
reduce need for assistance
PMA
don't bring annoyances
Patient
follow medical instructions
don't bring annoyances
reduce need for assistance

Control questions:

Which plans are activated in the following scenarios?

The patient is very accurate and does not miss any meal and medicine. Which plans are activated during a day? Plans: ____

The patient did eat for breakfast and lunch. Now its dinner time. What happens supposing that the patient will not eat it?
Plans: ____

If, answering to these questions you encountered any difficulty, please try now to correct the model.

7A

Questionnaire for example "Patient Monitoring Agent"

Post-modelling questions:

How much % of the time did you approximately spend in:

Re-reading the Tropos explaination	_____%
Reading & understanding the example	_____%
Modelling the example	_____%

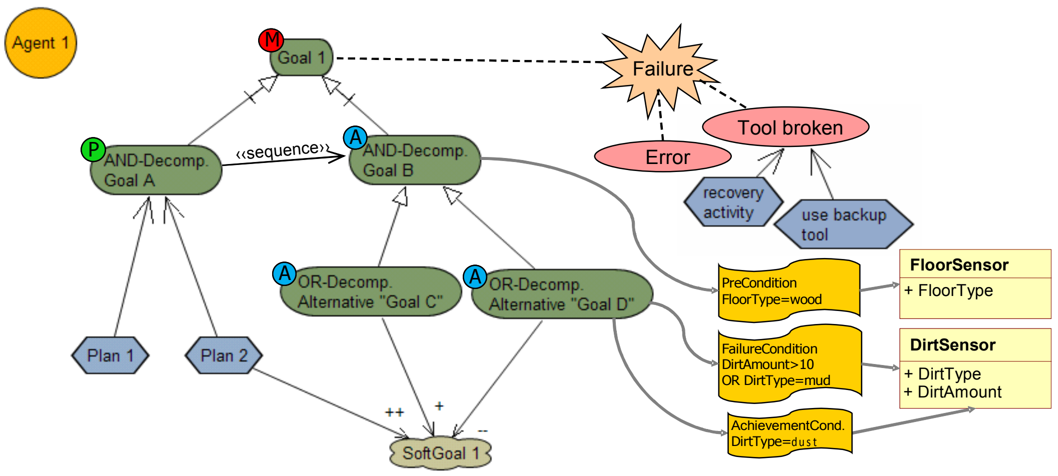
1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

- The explanation of the example was clear to me ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had no difficulties in modelling its requirements in a goal model ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had enough time for accomplishing the modelling task. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- Goal decomposition was very useful in this modelling task. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- The concepts of the modelling language were detailed enough to model the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had difficulties in modelling the user preferences with contributions to the softgoals. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- The effort of modelling seems too high for an efficient use of the methodology in practice. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- Suppose to be the programmer that has to detail and to implement the application. What do you think:
  - The obtained model is concrete enough to guide the programmers to an implementation respecting the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
  - The obtained model is too abstract to be able to properly guide the programmers to an implementation respecting the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

Additional questions (Tropos4AS modelling only") ↪

3B

Tropos4AS



Tropos4AS includes all TROPOS: **goals**, **softgoals**, **plans** (activities) to execute to reach goals; **AND/OR decomposition** of goals, **contribution** to softgoals (++, +, -, --).

Environment model: Defines the agent's perception of the environment

- **Environment entities**, represented as simplified UML classes, provide functionalities to sense and to act:
  - Entities used by the agent, with the functionalities to act and to sense (to evaluate conditions). e.g. Tropos Resources, devices in the system (e.g. *battery* with sensing functionalities *capacity*, *isCharging*,...) and outside (battery *charger*,...).
- Goals and plans are connected to environment artifacts through conditions:
  - **PreCondition**: the goal/plan can only be activated if it is true.
  - **CreationConditon**: activates the goal. Note: sub-goals are activated implicitly through the decompositions.
  - **AchievementCondition/MaintainCondition**: achieve / maintain the state.
  - **FailureCondition**: if true, the goal fails.

Extended goal model: Detail dynamics of goal achievement

- Model many alternatives: some could be more specialised to a specific context, while others would contribute less to the softgoals, but are more general. Characterise alternatives by contribution to softgoals and by modelling PreConditions.
  - If at run-time some goal or plan fails, less optimal alternatives will be selected!
  - At run-time, users can **change importance of softgoals**, giving more or less weight to softgoal contributions, thus possibly changing selection decisions.
- Goal types:
  - (A) **Achievement**: achieve the state one time.
  - (M) **Maintain**: maintain a state over time (agent tries to act each time it is not maintained).
  - (P) **Perform**: the goal to succeed to perform an activity (of the ones modelled).
- Relations between two goals:
  - $A \xrightarrow{\text{‹‹sequence››}} B$: Achieve goal A before activating goal B
  - $A \xrightarrow{\text{‹‹inhibits››}} B$: Goal B can not be active as long as goal A is active.

Failure model: Elicit missing requirements regarding the exceptional work flow

- Explicit a possible **Failure** (also from modelled failure conditions), correlating it to a the corresponding goal (if any), e.g. “the bike is unable to brake”
- Find one or more **Errors** that can be the cause of such a failure, e.g. “rear brake cable broken”.
- Model **recovery activities** (plans or pieces of goal models) to recover from an error, to prevent failure, e.g. “brake gently with front brake”, “goal to replace brake cable”.

4B

Example: Washing machine manager

System story

Summary: You have to develop an "intelligent" washing machine manager (WMM) software for a high-tech washing machine. The main aim of the stakeholders (both the producers and the users) is, undoubtedly, that it washes clothes. Doing this, the machine should achieve an appropriate level of cleanness, while being energy efficient.

The WMM will have to take various high-level decisions on the cleaning process, regarding heating, water use, detergent use, etc. We assume that the low level washing process is handled by a separate controller chip that takes the values from the WMM in input.

The machine has the following sensors: a water heat sensor, a weight sensor to measure how much the tumble is filled with clothes, and a sensor that reports the dirt level of the water in the tumble (on a scale from 0 to 9).

The user has the following ways to control washing:

• An "Energy saving" adjustment wheel to select the relative user preference in a range between 0 (less saving) and 1 (most saving) and a "Super Cleanness" adjustment wheel to select between 0 (less clean) and 1 (most clean). Suppose that turning these wheels reflects directly to the importance the user gives to the two respective softgoals "EnergyEfficiency" and "Cleanness" represented in the provided goal model.

• An On/Off knob for delicate clothes

Requirements: The WMM has the goal to **wash the clothes** respecting the user's settings. One sub-goal is to **dose the detergent appropriately**, depending on the weight sensor data the system should select to put one portion of detergent (<2 kg clothes) or two (>2 kg clothes).

Another sub-goal is to heat the water appropriately. Depending on the user's preferences represented by importance of the softgoals "EnergyEfficiency" and "Cleanness", the water should be heated accordingly, to 50°C (contributes more to "EnergyEfficiency") or to 90°C (contrib. more to "Cleanness"). If the "delicate clothes" knob is pressed, these two temperatures to *achieve* should be lowered to 30°, and 60°C.

Moreover, a sub-goal of the system will be to **ensure that the washing water is clean enough**. This should be achieved adapting the behaviour to the sensed dirt level of the water in the machine: the water should be recycled internally if it is not too dirty. Otherwise, the system should drain out the dirt water and refill with fresh water. Possibly, this decision should also be influenced by the position of the "Energy saving" wheel and the "Super Cleanness" wheel . During water draining and refilling, inhibit heating!

[suggestion: first try to complete the model until here]

During the washing process, there could arise several problems that may let **fail** the system's top-goal. What if the reported **water dirt level is very high after the washing process** and thus the clothes could not really be clean? Possibly the clothes should be soaked in fresh water for some time. If the "Super Cleanness" wheel is in position 1 (most clean)., the system should even start a new short washing cycle with one portion of detergent. Try to model this and imagine another possible error and activities to be done to prevent this failure!

Modelling Task

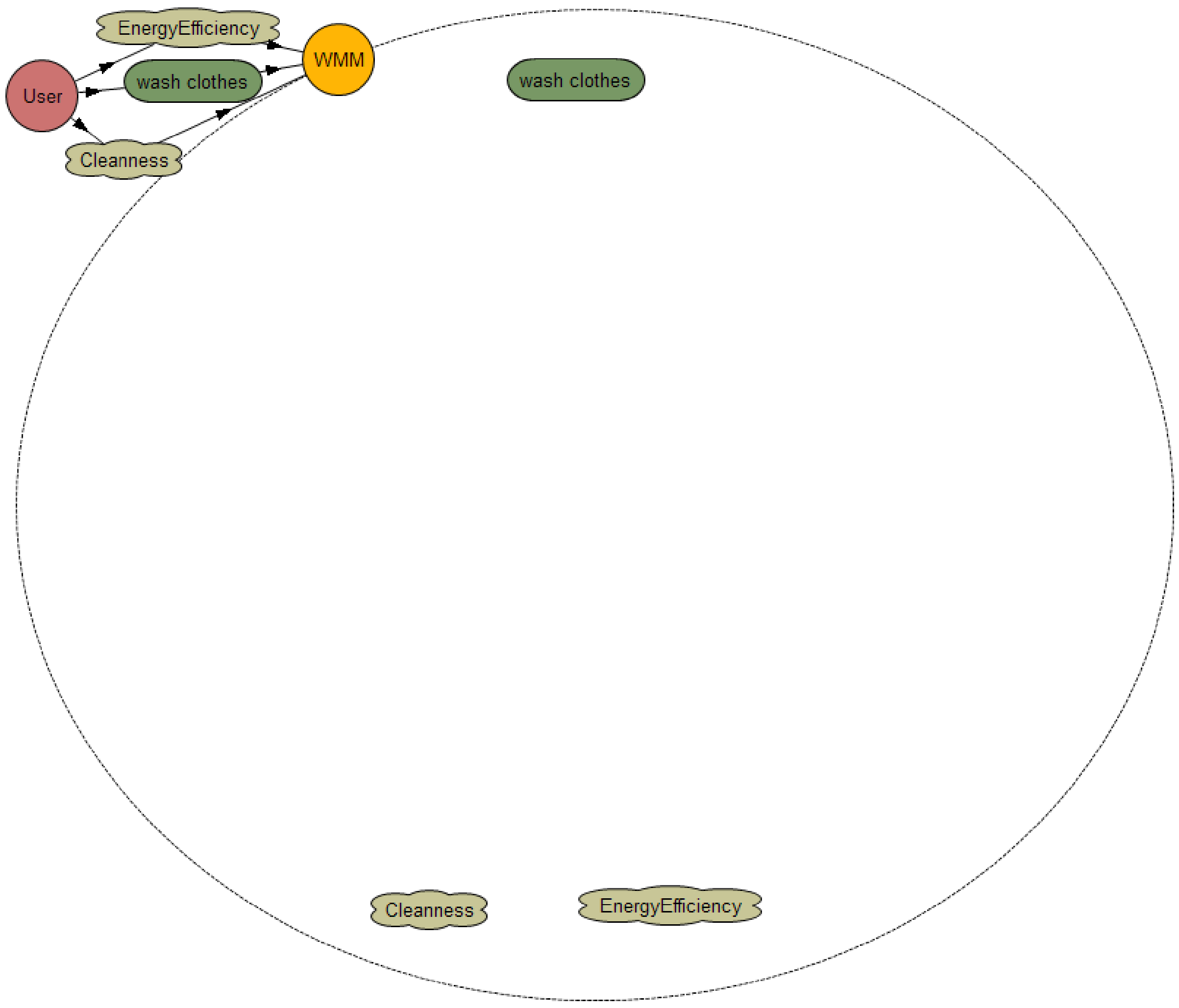
Try to model the requirements for the Washing Machine Manager as detailed as possible, following Tropos4AS, starting from the goal model of the actor "WMM" provided to you, which already contains the main goal and two softgoals. Draw the models by hand, following the modelling guidelines provided. Feel free to ask us questions on modelling constraints.

Model the example step by step following the system story. When modelling, try to freely interpret the requirements. There is no "right" or "expected" solution your models have to conform to. Remember to model contributions to the softgoals!

5B

Washing machine manager Tropos4AS ***model***

Start time __________ ***End time*** __________



Control questions:

Which plans are activated in the following scenarios?

The machine is fully filled with 4kg of delicate clothes and the “Optimal Energy” wheel is in “energy saving” position:

Plans: ____

Suppose that at the end of this washing cycle the water is very dirty. Which plans will be activated?

Plans: ____

If, answering to these questions you encountered any difficulty, please try now to correct the model.

Questionnaire for example “Washing Machine Manager”

Post-modelling questions:

How much % of the time did you approximately spend in:

Re-reading the Tropos4AS explaination	_____%
Reading & understanding the example	_____%
Modelling the example	_____%

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

- The explanation of the example was clear to me ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had no difficulties in modelling its requirements in a goal model ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had enough time for accomplishing the modelling task. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- Goal decomposition was very useful in this modelling task. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- The concepts of the modelling language were detailed enough to model the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- I had difficulties in modelling the user preferences with contributions to the softgoals. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- The effort of modelling seems too high for an efficient use of the methodology in practice. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
- Suppose to be the programmer that has to detail and to implement the application. What do you think:
    - The obtained model is concrete enough to guide the programmers to an implementation respecting the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5
    - The obtained model is too abstract to be able to properly guide the programmers to an implementation respecting the requirements. ❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

Additional questions (Tropos4AS only) ↪

8B

Questions for the Tropos4AS modelling task:

How much % of the modelling time did you approximately spend on

Goal modelling:	_____%	Environment&conditions modelling: _____%
Failure modelling:	_____%	(sum = 100%).

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

- In my opinion, enriching Tropos with conditions modelling is very useful for the scope of modelling adaptivity to the environment ❑ **1** ❑ **2** ❑ **3** ❑ **4** ❑ **5**

  Your thoughts about conditions modelling:____________________________________

  ____

- In my opinion, enriching Tropos with failure modelling is very useful for the scope of modelling adaptivity to the environment ❑ **1** ❑ **2** ❑ **3** ❑ **4** ❑ **5**

  Your thoughts about failure modelling: ____

  ____

1C

Comprehension study   Tropos vs. Tropos4AS

Example: Patient Monitoring Agent

Task:

Answer to the model comprehension questions on the next page, looking at the goal model (either Tropos or Tropos4AS) assigned to you. If the answers are not in the model, or you have difficulties extracting them, you can read them also in the system requirements. Remember that you have a tight, fixed time for the whole task: **12 minutes**! Afterwards, please respond to the questionnaire.

System requirements

You have to develop an "intelligent" patient monitoring agent (PMA) software. The main aim of this software is to ensure that a patient, which is usually auto-sufficient and does not necessarily need the daily presence of a care assistant, follows the medical instructions on eating meals and taking medicine. The system should reduce the need for human assistance (and thus, reduce the cost for health care), but it should also not bring annoyances to the patient's life.

The system should be installed in a "smart home" environment where it can access to reliable sensors that are able to measure if the patient takes the medicine and how often he eats (for the sake of simplicity, we suppose that such sensors exist). The flat is set up with loudspeakers in each room. It is also connected to the phone line for the system to request assistance from the care-assistant.

The PMA has the goal to ensure that the patient follows the medical instructions. This goal can be detailed to two sub-goals: eating meals regularly, and taking the medicine on schedule. To eat regularly, by the instructions of the doctor, the patient has to have at least two meals a day.

The patient has the possibility to eat for breakfast at 8AM, for lunch at noon and for dinner at 7PM. For each meal, starting from the specific time, for one hour, the patient should only be monitored to see if he eats autonomously. In this way the system does not bring any annoyances to accurate patients. Only after one hour the PMA should change its plan and gently remember the patient repeatedly, through the loudspeakers installed in the apartment. Obviously, with this the system contributes negatively to the requirement of not bringing annoyances, but still it contributes positively to a reduction of the need for human assistance. If the patient does not eat the meal also after one hour of repeated requests through the loudspeakers, the system should cease to do the requests and the respective plans fail. But only if the patient will no more be able to eat two meals in a day, assistance from the case-assistant should be requested.

The medicine should be taken daily in the evening, between 6PM and 8PM. Moreover, it is important that the patient does not eat after taking the medicine, and thus the medicine has to be taken after the last meal of the day. The system should ensure that the patient takes the medicine on within the given time interval and remember the patient for taking it, gently giving request through the loudspeaker system. If the medicine was not taken until 8PM, the care-assistant should be called for assistance.

If the patient is late with eating the dinner, he should skip it, to be able to take the medicine on time. However, if he did not eat both for breakfast and for lunch, this is not possible and thus the care-assistant has to be called also in this circumstance**.**

1C

Comprehension study Tropos vs. Tropos4AS

Example: Washing machine manager

Task:

Answer to the model comprehension questions on the next page, looking at the goal model (either Tropos or Tropos4AS) assigned to you. If the answers are not in the model, or you have difficulties extracting them, you can read them also in the system requirements. Remember that you have a tight, fixed time for the whole task: **12 minutes**! Afterwards, please respond to the questionnaire.

System requirements

You have to develop an “intelligent” washing machine manager (WMM) software for a high-tech washing machine. The main aim of the stakeholders (both the manufacturers and the users) is, undoubtedly, that it washes clothes. Doing this, the machine should achieve an appropriate level of cleanness, while being energy efficient. The WMM will have to take various high-level decisions on the cleaning process, regarding heating, water use, detergent use, etc. We assume that the low level washing process is handled by a separate controller chip that takes the values from the WMM in input. The machine has various sensors: a weight sensor to measure how much the tumble is filled with clothes, a sensor that reports the dirt level of the water in the tumble (on a scale from 0 to 9) and a water heat sensor.

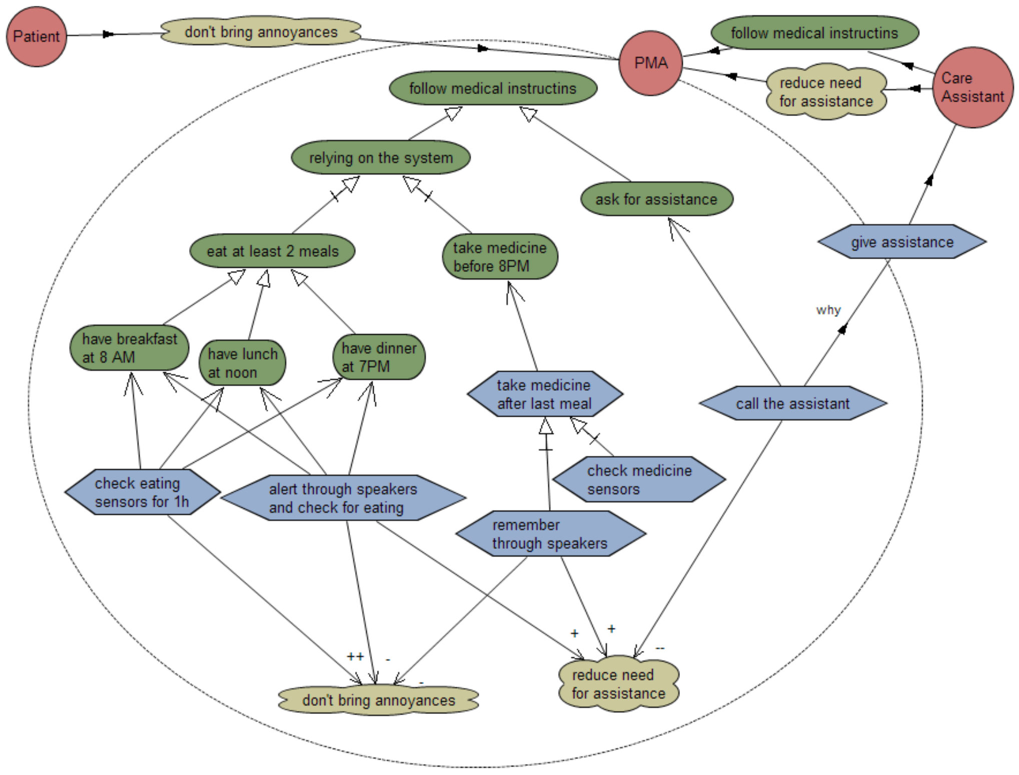
To control washing, the user can do various settings on the washing machine: An “Energy saving” adjustment wheel to select the relative user preference in a range between 0 (less saving) and 1 (most saving) and a “Super Cleanness” adjustment wheel to select between 0 (less clean) and 1 (most clean). Suppose that turning these wheels reflects directly to the importance the user gives to the two respective softgoals “EnergyEfficiency” and “Cleanness” represented in the provided goal model. Moreover, the machine has an On/Off knob for delicate clothes

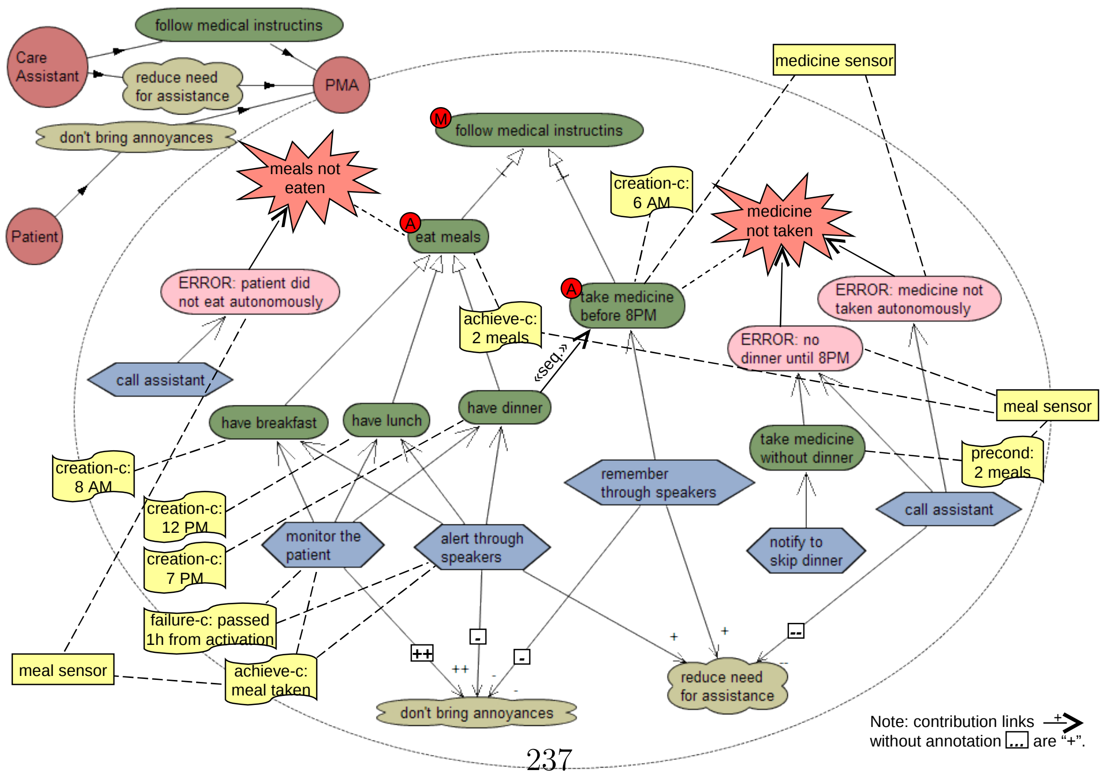
The WMM should wash the clothes, cleaning them, respecting the user's settings. One requirement is to dose the detergent appropriately, depending on the weight sensor data the system should select to put one portion of detergent (<2 kg clothes) or two (>2 kg clothes).

A further requirement for the system is to heat the water appropriately. Depending on the user's preferences represented by the position of the two wheels softgoals “EnergyEfficiency” and “Cleanness”, the water should be heated accordingly, to 50°C (contributes more to “EnergyEfficiency”) or to 90°C (contributes more to “Cleanness”). If the “delicate clothes” knob is pressed (before starting the washing cycle), the two temperatures to achieve should be lowered to 30°, and 60°C, respectively.

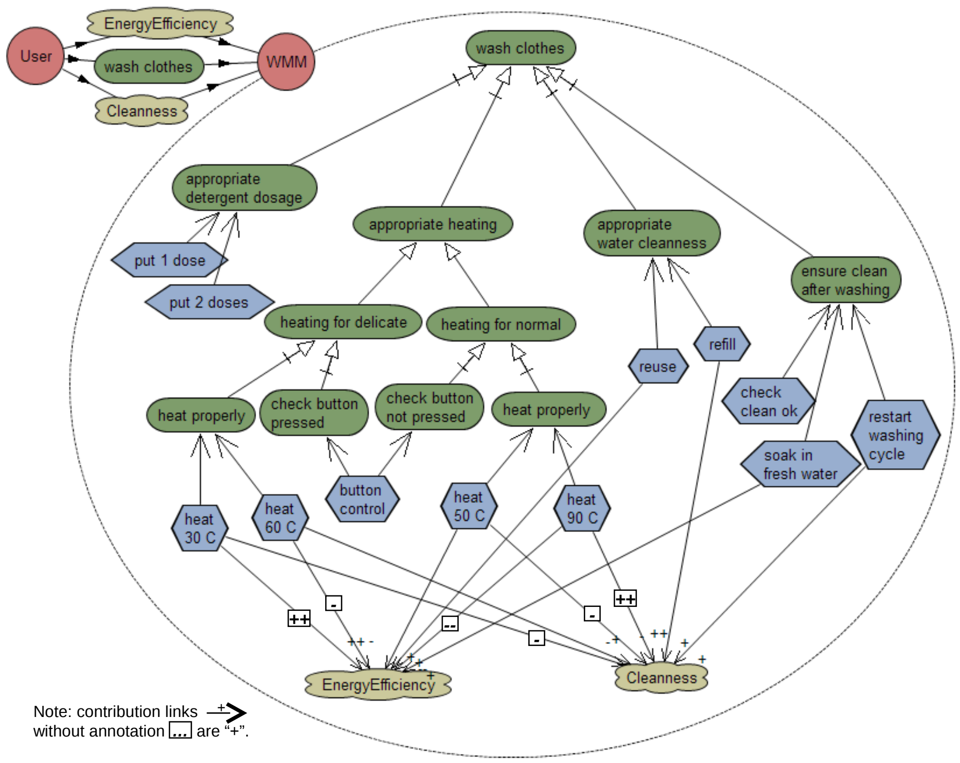
Moreover, the system should ensure that the washing water (which is typically recycled several times during the washing cycle) is clean enough. This should be achieved adapting the behaviour to the sensed dirt level of the water in the machine, thus the machine should drain out the dirt water and refill with fresh water if it is too dirty (at least a value of 5 on a scale from 1 to 10, measured by the water dirt sensor). However, this decision should also be influenced by the position of the “Energy saving” wheel and the “Super Cleanness” wheel. If energy efficiency is more important for the user than cleanness, the system should command the washing machine to recycle the water, up to a dirtiness threshold of 7. During water draining and refilling, heating should be inhibited, to avoid an unnecessary waste of energy.

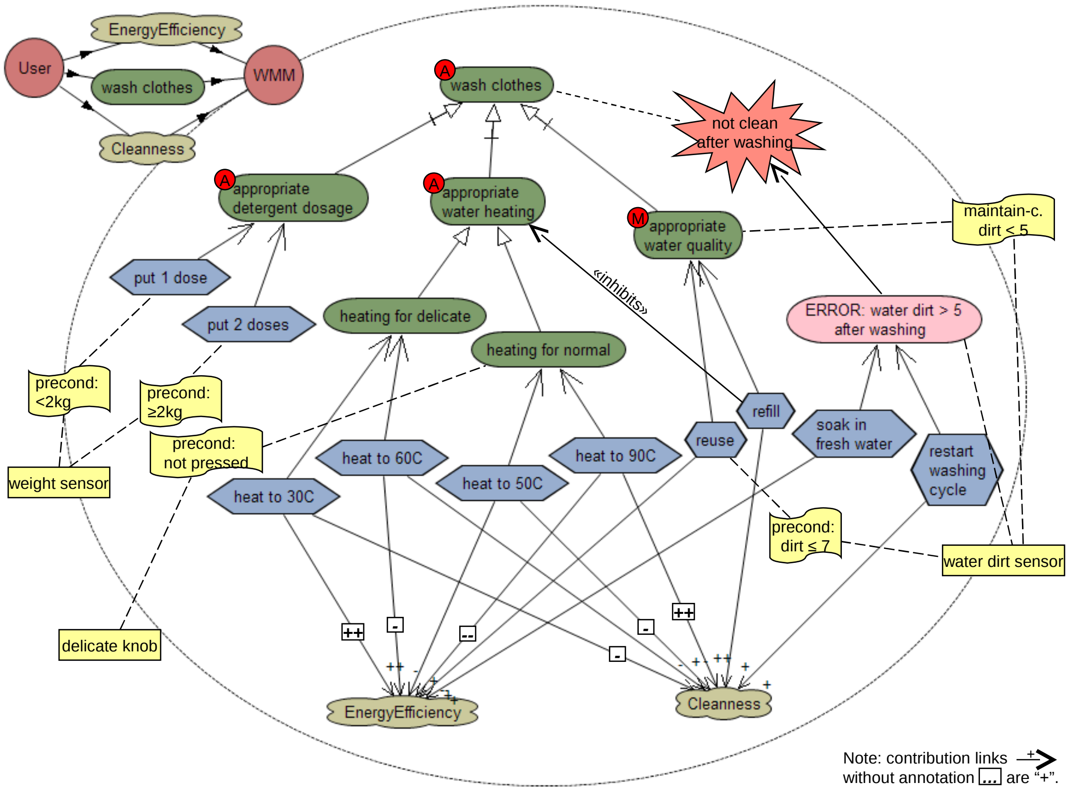
During the washing process, there could arise several problems that may let fail the system's main requirements. If, after the washing process, the water dirt level reported by the sensor is still very high (more than 5 on the scale from 1 to 10), the clothes are not yet clean enough. In this case, the clothes should be soaked in fresh water for some time, or, if cleanness is more important for the user than energy efficiency (i.e. the “Super Cleanness” adjustment wheel is set to a higher position), the system should even start a new washing cycle**.**





Note: contribution links $\xrightarrow{+}$ without annotation $\boxed{...}$ are "+".


EnergyEfficiency
User
wash clothes
WMM
Cleanness
wash clothes
appropriate detergent dosage
appropriate heating
appropriate water cleanness
ensure clean after washing
put 1 dose
put 2 doses
heating for delicate
heating for normal
reuse
refill
check clean ok
restart washing cycle
soak in fresh water
heat properly
check button pressed
check button not pressed
heat properly
button control
heat 30 C
heat 60 C
heat 50 C
heat 90 C
++
-
--
-
-
++
EnergyEfficiency
Cleanness
Note: contribution links without annotation are "+".


EnergyEfficiency
User
wash clothes
WMM
Cleanness
A wash clothes
not clean after washing
A appropriate detergent dosage
A appropriate water heating
M appropriate water quality
maintain-c. dirt < 5
put 1 dose
put 2 doses
heating for delicate
heating for normal
«inhibits»
ERROR: water dirt > 5 after washing
precond: <2kg
precond: ≥2kg
precond: not pressed
weight sensor
refill
reuse
soak in fresh water
restart washing cycle
heat to 60C
heat to 90C
heat to 50C
heat to 30C
precond: dirt ≤ 7
water dirt sensor
delicate knob
++
-
--
-
-
++
EnergyEfficiency
Cleanness
Note: contribution links without annotation are "+".

2C

PMA Model comprehension questions ***Tropos I***

Time: 12 minutes. Start: __________

1. In which occasions can the dinner be skipped?

____

____

2. Which activities will the system perform during the day, if a careful patient takes every meal at the scheduled time, and the medicine right after being prompted to by the system?

____

____

3. When exactly will the patient be remembered to take the medicine (after which patient's and/or system's actions)?

____

____

4. With which sensors does the system have to interact to get the necessary information?

____

____

5. In which circumstances will the system call the assistant? (please list all possibilities and write "why" (of the form e.g. "after breakfast, if the coffee was cold")

____

____

End time ____________

Post-Questionnaire (AFTER finishing the comprehension task)

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

- I had enough time for accomplishing the tasks **❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5**
- The comprehension questions were clear **❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5**
- I was able to extract the asked information from the goal model

  ❑ 1 nearly all ❑ 2 most ❑ 3 half ❑ 4 somewhat ❑ 5 nearly nothing
- I needed to search the information in the textual requirements

  ❑ 1 nearly all ❑ 2 most ❑ 3 half ❑ 4 somewhat ❑ 5 nearly nothing

THANK YOU!

2C

WMM Model comprehension questions ***Tr4AS II***

Time: 12 minutes. Start: __________

1. 3 kg of delicate, but very dirty clothes are put into the washing machine. Energy efficiency is much more important than cleanness.
   Which dosage and heat will the WMM system set?

2. With the clothes and settings from question 1, the water becomes dirtier and dirtier (i.e. dirty=1, dirty=2, ...) during the washing cycle. How will the system ensure an appropriate water quality? Give details!

3. What happens if cleanness is much more important than energy efficiency, and the water is still very dirty after the cleaning process? Suppose that this activity will fail. What will the system do?

4. Which sensor interfaces are needed?

5. With which user settings and conditions will the system restart a new washing cycle?

End time ____________

Post-Questionnaire (AFTER finishing the comprehension task)

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

- I had enough time for accomplishing the tasks **❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5**
- The comprehension questions were clear **❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5**
- I was able to extract the asked information from the goal model

  ❑ 1 nearly all ❑ 2 most ❑ 3 half ❑ 4 somewhat ❑ 5 nearly nothing
- I needed to search the information in the textual requirements

  ❑ 1 nearly all ❑ 2 most ❑ 3 half ❑ 4 somewhat ❑ 5 nearly nothing

THANK YOU!

Appendix B

Own Publications

The contributions of this thesis are supported by a set of publications carried out throughout this research work, with main contributions of the author. These have been published in several international conferences and workshops:

Mirko Morandini, Luca Sabatucci, Alberto Siena, John Mylopoulos, Loris Penserini, Anna Perini, and Angelo Susi. *On the use of the Goal-Oriented Paradigm for System Design and Law Compliance Reasoning.* In Proceedings of the 4th International i* Workshop (i* 2010) at CAiSE'10, Hammamet, Tunisia, June 2010, pp. 71-75.

Mirko Morandini, Frederic Migeon, Marie-Pierre Gleizes, Christine Maurel, Loris Penserini, and Anna Perini. *A Goal-Oriented Approach for Modelling Self-Organising MAS.* In 10th International Workshop on Engineering Societies in the Agents' World (ESAW 2009), Utrecht, November 2009, Springer, LNCS vol. 5881, pp. 33-48.

Mirko Morandini, Loris Penserini and Anna Perini. *Operational Semantics of Goal Models in Adaptive Agents.* In Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'09), Budapest, Hungary, May 2009, IFAAMAS, pp. 129-136.

Mirko Morandini, Loris Penserini and Anna Perini. *Modelling Self-Adaptivity: A Goal-Oriented Approach.* In 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Venezia, Italy, 20-24 October 2008. IEEE Press, pp. 469-470.

Mirko Morandini, Loris Penserini, Anna Perini. *Automated Mapping from Goal Models to Self-Adaptive Systems.* In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), demo session, L'Aquila, Italy, September 2008, IEEE Press, pp. 485-486.

Mirko Morandini, Loris Penserini and Anna Perini. *Towards Goal-Oriented Development of Self-Adaptive Systems.* In Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) at ICSE08, Leipzig, Germany, May 2008, ACM, pp. 9-16.

Loris Penserini, Anna Perini, Angelo Susi, Mirko Morandini, and John Mylopoulos. *A Design Framework for Generating BDI-Agents from Goal Models.* In Proceedings of the 6th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07), Honolulu, Hawaii, May 2007, IFAAMAS, pp. 610-612.

Mirko Morandini, Loris Penserini, Anna Perini and Angelo Susi. *Refining Goal Models by Evaluating System Behaviour.* In Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering (AOSE 2007), Honolulu, Hawaii, May 2007. In Agent Oriented Software Engineering VIII. LNCS Vol. 4951, Lin Padgham, Michael Luck editors. Springer, 2008, pp. 44-57.

Mirko Morandini, Duy Cu Nguyen, Anna Perini, Alberto Siena and Angelo Susi. *Tool-supported Development with Tropos: the Conference Management System Case Study.* In Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering (AOSE 2007), Honolulu, Hawaii, May 2007. In Agent Oriented Software Engineering VIII. LNCS Vol. 4951, Lin Padgham, Michael Luck editors. Springer, 2008, pp. 182-196.