WP1.2 - Definizione del processo di sviluppo

Università di Trento
ITC-irst

Abstract. This deliverable presents the results of the WP1-Task 1.2 and notably the definition of the methodology and the development process. It includes the analysis and comparison of diverse classical development process models and the tailoring of selected models to the methodology.

Document Identifier	Deliverable Dm.n
Project	MIUR-FIRB project RBNE0195K5 “Knowledge Level Automated Software Engineering”
Version	v1.0
Date	November 5, 2006
State	Final
Distribution	Public

Acknowledgements.

This document is part of a research project funded by the FIRB 2001 Programme of the “Ministero dell’Istruzione, dell’Università e della Ricerca” as project number RBNE0195K5.

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A..

Executive Summary

This deliverable presents the results of the WP1-Task 1.2 and notably the definition of the methodology and the development process. It includes the analysis and comparison of diverse classical development process models and the tailoring of selected models to the methodology.

In particular, it first introduces agent-oriented methodologies by considering both agent and multi-agent systems. Then, a new agent-oriented software development methodology, called Tropos, is presented and illustrated with the help of a case study. Finally, Tropos is confronted with the other existing agent-oriented methodologies, specifically with Agent OPEN.

Contents

Chapter 1

Introduction

A methodology aims to prescribe a set of concepts, methods and tools necessary for the commercial development of a software system. Prior to industry adoption, however, it is necessary for researchers to create that methodology. This has led to academic and industry researchers creating a large number of methodological approaches. A decade ago, there were estimated to be over a thousand methodological approaches to software development [Jay94] although these can be grouped into a much smaller number (around five) of software development approaches [IHK99]. To these can be added a sixth: that of agent-oriented (AO) methodologies i.e., methodological approaches suitable for the development of agent-oriented or agent-based software.

In parallel to the growth and availability of object-oriented (OO) systems development methodologies in the Nineties, we are now seeing the burgeoning of a number of innovative AO methodologies. However, in contrast to OO methodologies, the field is not industry-driven - most AO methodologies are supported by small teams of academic researchers. Based on an observation that the coalescence of groups of OO methodologies in the late 1990s led to an increased take-up by industry of the object-oriented paradigm for system development and project management, the idea is to encourage first the coalescence and collaboration between research groups and, then, hopefully, to more rapid industry adoption of AO methodological approaches. In other words, most AO methodologies are (at the time of writing) in an early stage and still in the first context of mostly "academic" methodologies for agent-oriented systems development; albeit that many of these methodologies have been tested in small, industrial applications. One purpose of this work is to identify those predominant and tested AO methodologies, characterize them, analyse them and seek some method of unification and consolidation with the hope that, in so doing, the community of scholars supporting AO methodologies will soon be able to transfer those innovative ideas into industry acceptance. This means mimicking the OO transition curve by seeking consolidation. One means of such consolidation is the use of a method engineering framework to create a repository of agent-oriented method fragments ([GHS05], [HSG05], [GCL+06], [BGHSW05], [GMO05], [GHSW04], [GMO04], [BGGM01], [GLWY02a], [GLWY02b], [MGK02], [GPM+01a], [DFM$^+$06], [MKFG05]).

From another point of view, agent oriented programming (AOP, from now on) is most often motivated by the need of open architectures that continuously change and evolve to accommodate new components and meet new requirements. More and more, software must operate on different platforms, without recompilations, and with minimal assumptions about its operating environment and users. It must be robust, autonomous and proactive. Examples of applications where AOP seems most suited and which are most quoted in literature [Wei99, WJ95] are electronic commerce, enterprise resource planning, air-traffic control systems, personal digital assistants, and so on.

To be qualified as an agent, a software or hardware system is often required to have properties such as autonomy, social ability, reactivity, and proactivity. Other attributes which are sometimes requested [WJ95] are mobility, veracity, rationality, and so on. The key feature which makes it possible to implement systems with the above properties is that programming is done at a very abstract level, more precisely, using a terminology introduced by Newell, at the *knowledge level* [New82]. Thus, in AOP, we talk of mental states, of beliefs instead of machine states, of plans and actions instead of programs, of communication, negotiation and social ability instead of interaction and I/O functionalities, of goals, desires, and so on. Mental notions provide, at least in part, the software with the extra flexibility needed in order to deal with the intrinsic complexity of the applications as mentioned before. The explicit representation and manipulation of goals and plans facilitates, for instance, a run-time "adjustment" of the system behavior needed in order to cope with unforeseen circumstances, or for a more meaningful interaction with other human and software agents.

1.1 Technical contribution

We are proposing a software development methodology, called *Tropos*, which will allow us to exploit all the flexibility provided by AOP (**[BGG$^+$04b]**, [GKMC05], [GKMP04b], [SPMG05], [BPG$^+$01b], [BPG$^+$01a], [PBG$^+$01], [GGMS02]). In a nutshell, the two key and novel features of Tropos are the following.

1. The notion of agent and all the related mentalistic notions are used in all the software development phases, from the first phase of early requirements analysis down to the actual implementation. Particularly, we focus on BDI (Belief, Desire, and Intention) agent architectures [RG91].

2. A crucial role is given to the early requirements analysis that precedes the prescriptive requirements specification. We consider much earlier phases of the software development than the phases supported by other agent or object oriented software engineering methodologies. As described below, this move is crucial in order to achieve our objectives.

The idea of paying attention to the activities that precede the specification of the prescriptive requirements, such as understanding how the intended system would meet the organizational goals, is not new. It has been first proposed in requirements engineering, see for instance [DvLF93, Yu95a], and in particular by Eric Yu with his *i** model. Applied in various application areas, including requirements engineering [Yu93], business process reengineering [YM96], and software modeling processes [YM94], the *i** model offers actors, goals and actor dependencies as primitive concepts [Yu95a]. The main motivation underlying this earlier work was to develop a richer conceptual framework for modeling processes which involve multiple participants (both humans and computers). The goal was to have a more systematic reengineering processes. One of the main advantages is that, by doing an earlier analysis, one can capture not only the *what* or the *how*, but also the *why* a piece of software is developed. This, in turn, supports a more refined analysis of the system dependencies and, in particular, for a much better and uniform treatment, not only of the system's functional requirements, but also of the non-functional requirements (the latter being usually very hard to deal with).

Neither Yu's work, nor, as far as we know, any of the previous work in requirements analysis was developed with AOP in mind. The application of these ideas to AOP, and the decision to use mentalistic notions in all the phases of analysis, has important consequences. When writing agent oriented specifications and programs, one uses the same notions and abstractions used to describe the behavior of the human agents, and the processes involving them. The conceptual gap from *what* the system must do and *why*, and *what* the users interacting with it must do and *why*, is reduced to a minimum, thus providing (part of) the extra flexibility needed to cope with the applications' complexity.

Indeed, the software engineering methodologies and specification languages developed for Object-Oriented Programming (OOP) support only the phases from the architectural design downwards. At that moment, any connection between the intentions of the different agents (human and software) cannot be explicitly specified. By using UML, for instance, the software engineer can start with the use case analysis (possibly refined with activity diagrams) and then move to the architectural design. Here, the engineer can do static analysis using class diagrams, or dynamic analysis using, for instance, sequence or interaction diagrams. The target is to reach the detail of the abstraction level of the actual classes, methods and attributes used to implement the system. However, applying this approach and the related diagrams to AOP, the software engineer misses most of the advantages coming for the fact that in AOP one writes programs at the knowledge level. It forces the programmer to translate goals and the other mentalistic notions into software level notions, for instance classes, attributes and methods of class diagrams. The consequent negative effect is that the former notions must be reintroduced in the programming phase. The work on AUML [BMO01, OPB00], though relevant in that it provides a first mapping from OOP to AOP specifications, is an example of work suffering from this kind of problem.

In another part of this work, we make the (common) assumption that adding support for agent concepts into an existing object-oriented methodological approach is feasible and useful. We begin with the OO approach offered by the OPEN Process Framework or OPF [FHS02b]. Although some basic agent concepts have recently been added [DHS03b] to create "Agent OPEN", Agent OPEN still lacks the sophisticated support for agents necessary to provide complete support for agent-oriented software development (AOSD).

Although it would be possible to use a combination of methodologies (e.g., here, OPEN complemented by Tropos), it is not practical for industry, which preferably requires a single, coherent and integrated "package" to support application development. The first AO methodology, whose integration with OPEN AOSD methodology is analyzed, is Tropos [BGG$^+$03], which stresses the need for agent-orientation in early requirements engineering — topics not already addressed in the OPEN literature (**[HSGB03a]**, [HSGB03b]).

1.2 Plan of the deliverable

As an introduction to agent-oriented (AO) methodologies, we first describe the characteristics of both agents and multi-agent systems (MASs). This leads to a discussion of what makes an AO methodology that can be used to build an MAS.

Next we introduce and motivate the *Tropos* methodology,[1] for building agent oriented software systems. Tropos is based on two key ideas. First, the notion of agent and all the related mentalistic notions (for instance, goals and plans) are used in all phases of software development, from the early analysis down to the actual implementation. Second, Tropos covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents. The methodology is illustrated with the help of a case study. The Tropos language for conceptual modeling is formalized in a metamodel described with a set of UML class diagrams.

Integrating agent concepts into existing OO methodologies has resulted in several agent-oriented methodologies, one of which is Agent OPEN. In the last section, we evaluate the existing Agent OPEN description against ideas formulated within Tropos.

[1]From the Greek "tropé", which means "easily changeable", also "easily adaptable".

Chapter 2

State of the art of AOSE

2.1 Agents and Multi-Agent Systems

Defining agents is not straightforward. There are many opinions, some of which you will see reflected in [HSG05]. The key characteristics are widely understood to be highly autonomous, proactive, situated and directed software entities. Other characteristics such as mobility are optional and create a special subtype of agent; whereas some characteristics cannot be used as determining factors since they are really grey shades of a scale that encompasses both objects and agents.

In this context, an autonomous agent is one that is totally independent and can decide its own behaviour, particularly how it will respond to incoming communications from other agents or objects. A proactive agent is one that can act without any external prompts. However, it should be noted that this introduces some problems since there is also a large literature on purely *reactive* agents which would not be classified as agents with this categorization. In reality, most agents being designed today have both proactive and reactive behaviour. Balancing the two is the key challenge for designers of agent-oriented software systems.

The characteristic of situatedness means that agents are contained totally within some specific environment. They are able to perceive this environment, be acted upon by the environment and, in turn, affect the environment. Finally, the directedness means that agents possess some well-defined goal(s) and their behaviour is seen as being directed towards effecting or achieving that goal.

Comparison with objects is often made. Some see agents as ”clever objects” or ”objects that can say no”. This means that a hybrid agent+object system is entirely feasible. Others see agents at a much higher level of abstraction (e.g. [MCD$^+$01]) much in the same way that OO specialists view components at a similar more granular level. Indeed, it is still unresolved as to how the scale of objects, components and agents are matched and to what extent hybrid object/component/agent systems are feasible.

Some consequences of these high level definitions are that agents participate in decision-making cycles, sometimes labelled as "perceive-decide-act" cycles. To achieve this, we have to consider other lower-level characteristics such as the roles that agents play, the metaphor of the agents having a mental state, including the possession of skills and responsibilities, aptitudes and capabilities. When considering their interactions via perceptions and actions with other agents and the environment we introduce notions of perceptions, actions, agent communication languages. Negotiating skills involve the consideration of contract nets, auction strategies and the issues of competition versus cooperation.

Defining a Multi-Agent System is also not straightforward. However, almost all the definitions given in the literature conceive a MAS as a system composed of cooperative or competitive agents that interact one another in order to achieve personal or common goals. From the software engineering point of view, one of the most important characteristics of a MAS is that the set of agents is generally not given at design time, but at run time. This basically means that in practice MASs are based on open architectures that allow new agents to dynamically join and disjoin the system.

A MAS contains agents (and not, say, objects), consequently it inherits some of their characteristics. Thus key characteristics of an MAS can be said to be autonomy, situatedness, proactivity and sociality. This leads to a set of desirable high level characteristics [MCD$^+$01] of adaptiveness, flexibility, scalability, maintainability and likely emergent behaviour. While these are said to be "desirable" characteristics, it is perhaps the last of this list that causes most concern. Emergence is usually linked to Complex Adaptive Systems (CAS) theory. Although there are many shades of definition of what is meant by emergence in the CAS community, the general interpretation is that an emergent behaviour is one which cannot be predicted by inspection of the individual parts. This means that it is not visible from a bottom up analysis and, arguably therefore, must be considered at the system level. Since this sort of emergent behaviour is generally encouraged, allowing it to emerge unconstrained and unplanned is clearly dangerous in certain circumstances (whilst beneficial in others). To alleviate this concern of an uncontrolled and uncontrollable agent system wreaking havoc, clearly emergent behaviour has to be considered and planned for at the systems level using top-down analysis and design techniques. This is still an area that is largely unknown in MAS methodologies. Adelfe starts along this path with their consideration of adaptive MAS (see [HSG05, Chapter 7] for further details) in which agents that permanently try to maintain cooperative interactions with others.

Many AO methodologies (e.g., Gaia and Tropos) use the metaphor of the human organization (possibly divided into sub-organizations) in which agents play one or more roles and interact with each other. Human organization models and structures are consequently used to design MAS (see for instance the use of architectural patterns in Tropos or the organization models in MAS-commonKADS). Concepts like role, social dependency and organization rules are used not just to model the environment in which the system will work, but the system itself. Given the organization nature of a MAS, one of the most important activity in AO methodology results the definition of the interaction and cooperation models, that capture the social relationships and dependencies between agents and the roles they play within the system. Interaction and cooperation models are generally very abstract and they are concretized implementing interaction protocols in later phases of the design.

2.1.1 Developing multi-agent systems

Although the Agent-Oriented Programming (AOP) paradigm has been introduced more than twenty years ago by Yoav Shoam in his seminal work [Sho93], still there are no AO languages used in practice for developing MAS. Many tools have been developed in last years to support the implementation of agents and multi-agent systems, but still no one is based on a proper agent-oriented langue. An interesting and very long list of agent tools is available at the AgentLink web site (http://www.agentlink.org). Current agents developing tools are mainly built on top of java and use the object-oriented paradigm for implementing software. The methodologies presented in this book do not have as main focus the implementation phase, however many of them give some indications how to do that. The most used developing tools are JACK ([Cob00b]) and JADE ([JAD]).

JACK is a commercial agent-oriented development environment built on top and fully integrated with Java. It includes all components of the Java development environment as well as offering specific extensions to implement agent behavior. JACK provides agent-oriented extensions to the Java programming language and source code is first compiled into regular Java code before being executed. In JACK a system is modeled in terms of agents defined by of capabilities, which in turn are defined in terms of plans (set of actions), events, beliefs, and other capabilities.

JADE (Java Agent DEvelopment Framework) is a free software framework fully implemented in Java language. It allows for the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications and through a set of graphical tools that supports the debugging and deployment phases. The agent platform can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another one, as and when required.

2.2 What is an AO methodology?

While there is much debate on the use of terminology in various subcultures of information systems and software engineering, it can be generally agreed (e.g. [RPB99]) that a ”methodology” has two important components: one which describes the process elements of the approach and a second which focusses on the work products and their documentation. The second of these is more visible in the usage of a methodology, which is why the OO modelling language UML (OMG, 1999, 2004) is so frequently (totally incorrectly) equated with ”all things OO” or even described as a methodology! A modelling language such as this or its agent-focussed counterpart of AUML ([OVDPB00]) offers an important contribution but has limited scope within the context of a methodology. The emphasis placed on the product side of a methodology tends to vary between different authors, as will be seen in the remainder of this book. Some use UML/AUML and others eschew this as being inadequate to support the concepts of agents as described in Section 2, introducing instead their own individualistic notation. When UML style diagrams are used, it is assumed that the reader has some familiarity with this kind of graphical notation[1] ; otherwise, each author fully defines the notational set of icons being used in that particular methodological approach.

Although these two main components (process and product support) are generically agreed upon, it is possible to elaborate a little more since there are issues of people, social structures, project management, quality issues and support tools. These can be reconciled within the process domain along with concerns about metrics and standards, organizational procedures and norms and, if possible, all underpinned by a metamodel and ontology (e.g. [HS95]; [RP96]).

Any methodology also needs to contain sufficient abstractions to fully model and support agents and MASs - arguably, simple extensions of OO methodologies are too highly constrained by the sole focus on objects. Thus, an AO methodology needs to focus on an organized society of agents playing roles within an environment. Within such an MAS, agents interact according to protocols determined by the agents’ roles.

We should also ask what it means for a methodology to be ”agent-oriented” in the sense that we talk of an OO methodology in the context of object technology. In this case, however, object technology has two foci. In an OO methodology, we use OO concepts to describe the methodology, which, in turn, can be used to build object-oriented systems. In contrast, when we speak of an AO methodology, we generally do not mean a methodology that is itself constructed on agent-oriented principles but merely one that is oriented towards the creation of agent-based software. Thus all the chapters, except one, follow this ”definition”. Only Tropos (described in detail in Chapter 2) uses ”agent think” in its very derivation.

2.3 Genealogy of methodologies

Agent-oriented methodologies have several roots. Some are based on ideas from AI, others as direct extensions of existing OO methodologies, whilst yet others try and merge the two approaches by taking a more purist approach yet allowing OO ideas when these seem to be sufficient. Figure 1 shows these lineages and influences in what might be called a genealogy of the ten AO methodologies discussed in this book.

Several methodologies acknowledge a direct descendancy from full OO methods. In

[1]If not, a useful introduction to be found in [HSU00]

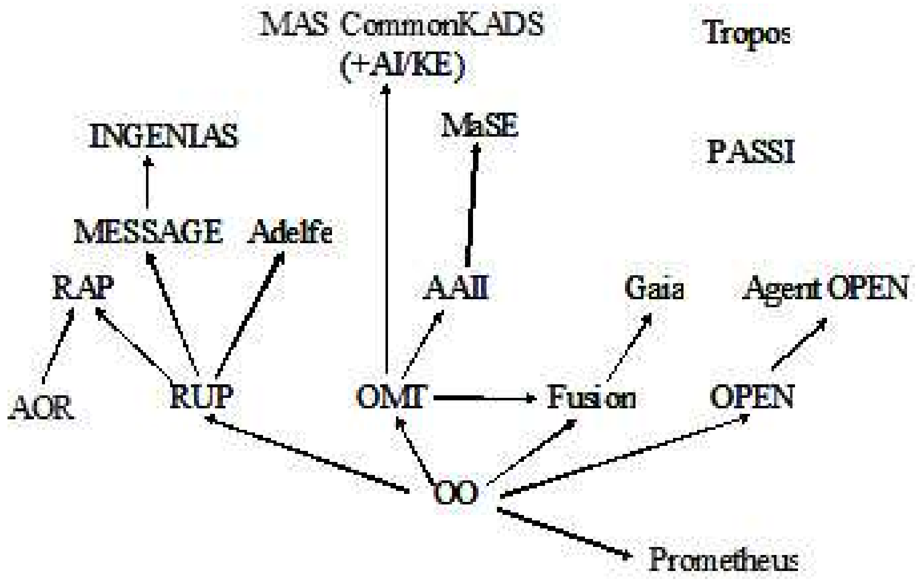


Figure 2.1: Genealogy of methodologies

particular, MaSE ([DeL99a]; [WD00]) acknowledges influences from [KMJ96] as well as an heredity from AAII ([KGR96]) which in turns was strongly influenced by the OO methodology of Rumbaugh and colleagues called OMT ([RBP+91]). Similarly, the OO methodology of Fusion ([CAB+94]) was said to be highly influential in the design of Gaia ([WJK00a]; [ZJW03]). Two other OO approaches have also been used as the basis for AO extensions. RUP ([Kru99]) has formed the basis for Adelfe ([BGPG02a]) and also for MESSAGE ([CCG+01]), which, in turn, is the basis for INGENIAS ([PGSF05]) and, more recently, for RAP ([WT05]), the last of these also being a direct descendant of AOR ([Wag03]). Secondly, the OPEN approach to OO software development has been extended significantly to support agents, sometimes called Agent OPEN ([DHS03a]). Finally, Prometheus, although not an OO descendant, does suggest using OO diagrams and concepts whenever they exist and are compatible with the agent-oriented paradigm.

Somewhat different is the MAS-CommonKADS methodology ([IGGV96], [IGGV98]). This is the only solidly-AI-based methodology discussed in this book yet also claims to have been strongly influenced by OO methodologies, notably OMT.

Then there are the methodologies that acknowledge no direct geneaological link to other approaches, OO or AO. Those discussed in this book are Tropos ([JMJ02]; [GKMP04a]; [BGG+04a]) and PASSI ([CP02]; [BC02]). Tropos has a significant input from i* ([Yu95b]) and a distinct strength in early requirements modelling, focussing as it does on describing the goals of stakeholders that describe the ”why” as well as the more standard support for ”what” and ”how”. This use in Tropos of the i* modelling language gives it a different look and feel from those that use Agent UML (a.k.a. AUML: [OVDPB00]) as a notation. It also means that the non-OO mindset permits users of Tropos to take a unique approach to the modelling of agents in the methodological context. Other approaches not covered in this book include Nemo ([Hug02]), MASSIVE ([Lin99]), Cassiopeia ([CDB96]; [CD98]) and CAMLE ([SZ04]) - although in CAMLE there are some parallels drawn between its notion of ”caste” and the concept of an OO class as well as some connection to UML’s composition and aggregation relationships.

Further comparisons of these methodologies are undertaken in [HSG05, Chapter 12], which complements and extends earlier framework-based evaluative studies of, for instance, [CR02], [DW04], [SS04] and [TLW04].

2.4 Common terms/concepts used in the AO methodologies

Agent terminology is not universally agreed upon. Nevertheless, there is sufficient agreement that makes it worthwhile for us to summarize commonly agreed terms here. Bear in mind, however, that each methodological approach may treat these terms slightly differently - as will be pointed out when necessary.

Agents are often contrasted with objects and the question raised ”What makes an agent an agent and not an object” - particularly difficult when one considers the concepts of ”active objects” in an OO modelling language such as the UML.

The novelty of agents is that they are proactive (as well as reactive), have a high degree of autonomy and are situated in and interact with their environment ([ZJW01]), which is sometimes considered simply as a *resource*. This introduces (beyond objects) issues to do with not only the environment itself but also the direct interactions between the agent(s) and their environment. Thus interfaces are as or more important in agents than in objects and, particularly, object-implemented components. When an agent perceives its environment, perhaps by means of a *sensor* of some kind, it is common to model that in terms of *percepts*. When an agent interacts with its environment in order to make a change in the environment, it is called an *action*. The mechanism by which this is accomplished is often called an *effector* (see [HSG05, Chapter 6]).

Agent behaviour can be classified as *reactive* or *proactive*. A reactive agent only responds to its environment. These changes are communicated to the agent as *events* (although events also occur as a direct result of messages sent from other agents or indeed sent internally to the agent). Thus changes in the environment have an immediate effect on the agent. The agent merely reacts to changing conditions and has no long term objectives of itself. In contrast, a proactive agent has its own objectives. These are usually

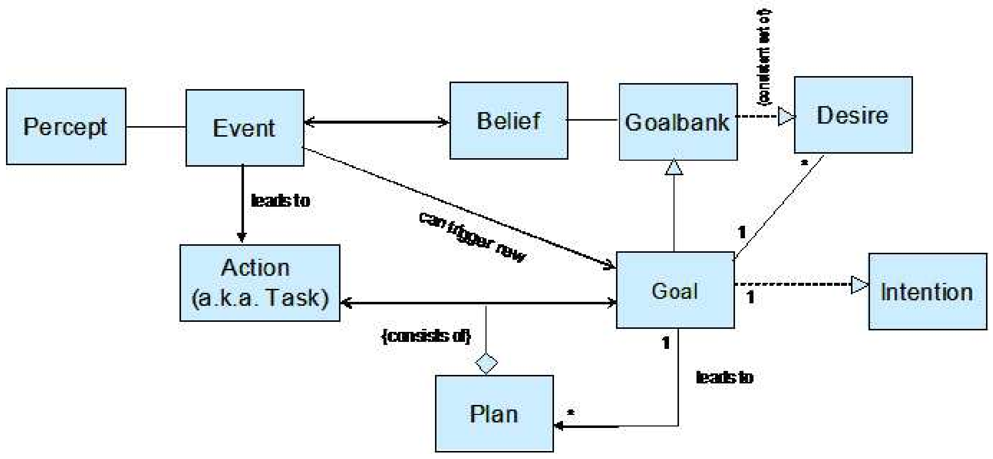


Figure 2.2: Metamodel fragment showing, *inter alia*, the links between goals and tasks

represented as one or more *goals* (e.g. [DLF93]; [KZ98]; [GKMP04a]). To achieve a goal, it is usual to construct a *plan* and then to execute that plan by means of process elements known as *actions* (or often as *tasks*) (Figure 2). In reality, many agents are designed as hybrid agent, possessing both reactive and proactive characteristics. The challenge then is for the designer to balance these two very different behaviours in order to create an overall optimal behaviour.

One well-known agent architecture that reflects these notions is the BDI (Beliefs, Desires and Intentions) architecture of [RG95] and [KGR96]. [WPH01] summarize this architecture in terms of three "abstraction layers" called philosophical (renamed here as psychological), theoretical and implementation (Table 1). Beliefs, Desires and Intentions are seen as high level, abstract, external characteristics. These three characteristics are then mapped through to the design or model layer. In particular, we note in Table 1 that Beliefs represent the agent's knowledge at various granularity levels, while Desires, which represent heterogeneous objectives, possibly including some conflicts, map to Goals (now a consistent set of objectives) within the agent and Intentions are mapped to Committed Goals (a coherent subset of goals with no conflicts or contradictions). In this table (from [HSTD05]), Plans are included specifically to account for the "how" element not originally included in the original BDI descriptions. Typically, each goal would have a link to at least one plan. These ideas have been described by [HSTD05] by a metamodel fragment as shown in Figure 3.

The BDI model is one description of the internal structure of an agent. However, agents are social entities and thus optimal performance is more likely from a cluster of

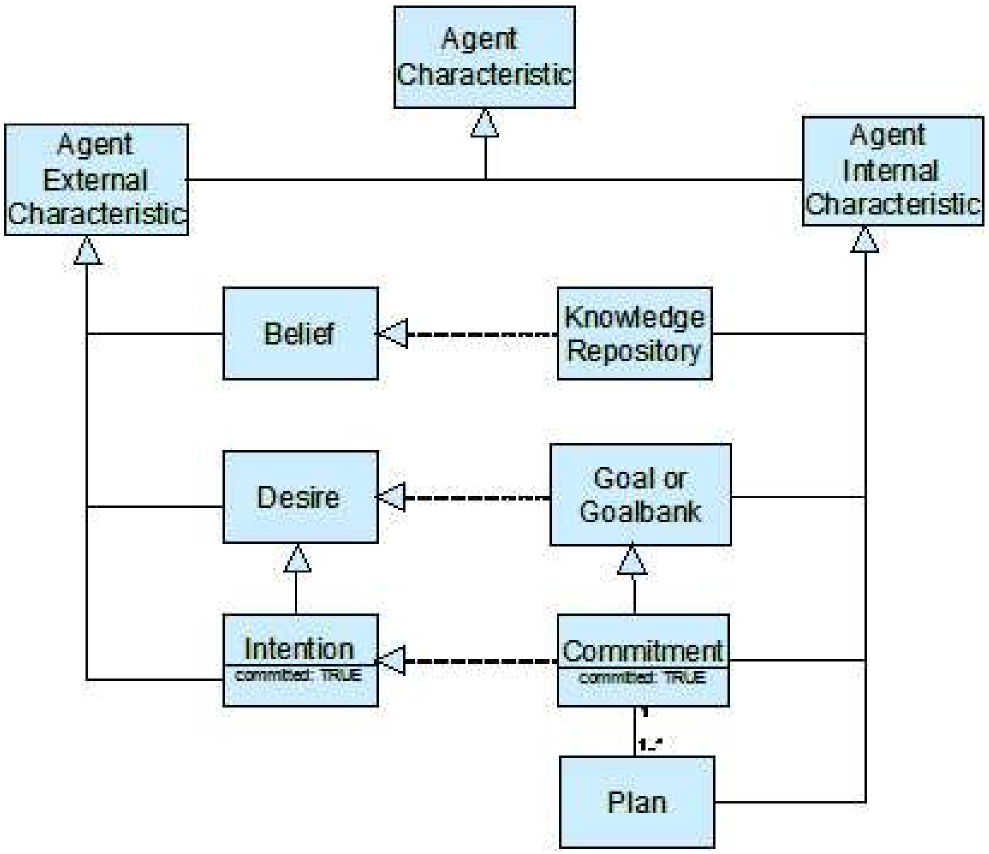
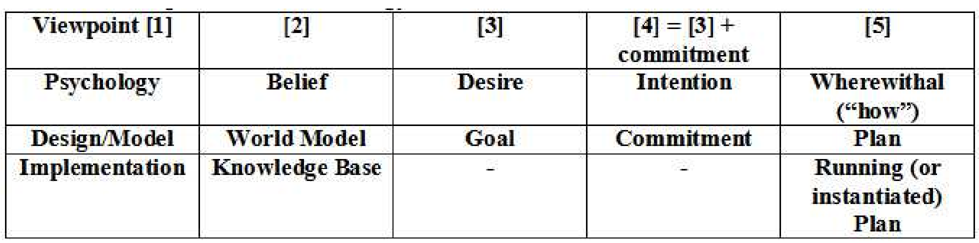


Figure 2.3: Metamodel of concepts used in the BDI architecture (after [HSTD05])

agents. This is an MAS or multi-agent system. The methodologies in this book are aimed at building MASs and not single agent systems. A metaphor that is often used is that of the (human) organization, in which the participating agents are said to exhibit *social* behaviour. Theories developed for human organizations are often usefully applied, such as that of social commitment ([YS02]; [CS98]) and social norms ([CDJT00]). Within this social environment, as with humans, agents are seen to form teams ([CL91]) in which agents play roles ([Ken00]). In fact, the notion of a role, while supported in some OO approaches (e.g. [FHS02a]), is a major differentiating factor for agents. In fact, in the comparative surveys discussed above and further in [HSG05, Chapter 12], one common, major classification axis is whether the AO methodology is a role-based one or not.

If agents are operating in a communal workplace, then clearly they need to communicate even more urgently than do objects. While objects are passive, reacting only to in-



Viewpoint [1]	[2]	[3]	[4] = [3] + commitment	[5]
Psychology	Belief	Desire	Intention	Wherewithal (“how”)
Design/Model	World Model	Goal	Commitment	Plan
Implementation	Knowledge Base	-	-	Running (or instantiated) Plan

Figure 2.4: Table 1 Relationships between terminology of BDI model (after [HSTD05])

coming messages, agents have the ability through their autonomy and proactive behaviour to initiate messages. Agent-to-agent communication is a major research area in MAS behaviour. We note here that the key issues are how to describe agent messages in terms of *interaction* and *communication* protocols, perhaps using a formal, mathematically-based language.

An MAS clearly contains many agents within the contextual environment. As well as inter-agent communication, we need to recognize that within an MAS, agents need to both compete and cooperate. Although essentially selfish in their autonomy, agents act like humans: sometimes aiming to fulfil their own goals at the expense of all other agents/humans but mostly in a more social structure in which it is recognized that collaboration and sharing of work is mutually beneficial as well as individualistically profitable. Thus the notion of agents, organized to work within a social structure, is also a very strong driver in AO methodologies (e.g. [ZJW01]). Indeed, some of the methodologies discussed in this book argue that their main differentiator is that they address these issues of social structure as a top priority, perhaps downplaying the discussions about agent infrastructure and whether models such as BDI are optimum.

Moving from agent infrastructure and communication to process elements, we should briefly outline the use of various terms such as lifecycle, analysis, design and implementation. In the 1970s and 1980s, these terms were generally understood and related to organizational structure within software development companies, particularly those using the waterfall approach. This described a number of ”phases”, the totality of which was called the SDLC or software development life cycle. However, the advent of object technology and OO software development methodologies throughout the 1990s led to an avoidance of these terms on the macro-scale.

Analysis is equated to the understanding of something already in existence, sometimes labelled ”discovery”. It relates to the problem space. In contrast, design is considered as part of the solution space in which various possible ”answers” are considered. Thus it could be labelled ”invention” ([Boo94]). The argument is that these do not occur sequentially on a timescale of months but in normal human cognition on a timescale of seconds and highly iteratively. Together, these two obsolescent phases were frequently called ”modelling” in OO methodologies. This leads to a modelling language such as UML having an equivalent scope i.e. ”analysis” and ”design”. This, in turn, often leads to confusion since clearly the ”modelling phase” is a lengthy one in which initially there is more analysis than design whereas towards the end of the phase there is more design than analysis going on. The use of any particular technique or modelling notation thus shifts in balance from those more useful for ”discovery” to those more focussed on the solution space. Notwithstanding, it is sometimes useful to re-introduce the analysis and design terms but not as straitjackets for two lengthy sequential, non-iterative phases, but simply to remind the reader and user of the methodology of the shifting balance described above, thus permitting discussion of more ”analysis-type/design-type” techniques under the banner of an ”analysis/design phase”.

An emerging framework that is hinted at in some chapters is MDA (Model-Driven Architecture). This is a fairly recent initiative of the Object Management Group (OMG) which attempts to supply an architectural infrastructure to SDLC by identifying a Platform-Independent Model (PIM) which avoids any assumptions about operating system, programming language, hardware etc. This PIM is then translated (finally the aim is to do this automatically) to a Platform Specific Model or PSM ([KWB03]). The applicability of this to AO methodologies awaits to be seen.

Chapter 3

Tropos, the KLASE proposed methodology

3.1 Towards Knowledge-Driven Peer-to-Peer computing

In the Peer-to-Peer (P2P) model computational peers interoperate in a completely decentralized distributed environment, providing to and requesting from each other data and/or services. Peers are largely autonomous in what and how they store in their local (knowledge) bases, in what data and services they provide to other peers, in what other peers they ”talk to”, etc. Peers come and go, make spontaneous acquaintances with other peers, and, eventually, drop them. Peers collaboratively process user requests, and the overall performance of the network emerges from local point-to-point interactions of (all) peers on the network.

P2P applications cover a number of domains such as file sharing (e.g., Kazaa), distributed computing (e.g., SETI@Home), collaborative networking (e.g., Groove), and instant messaging (e.g., ICQ). However, most of the P2P systems are hybrid - peers, (mainly) for discovery purposes, rely on a centralized resources indexing server. An example of a totally decentralized P2P application is Gnutella , which is a file sharing application. In file sharing applications, all peers rely on the same schema that describes the content of the files they share. What here needs to be underlined is that, although technologically distributed, P2P (data sharing) systems are at least conceptually centralized as far as they have to assume some shared semantics in order to exchange data meaningfully (e.g. names and meaning of categories). For a comprehensive overview of the P2P technology and applications see, for instance, [Milojicic 02].

According to some studies carried out at Gartner, P2P has passed its ”peak of inflated expectations”, and now true technology’s applicability, risks and benefits need to be understood . At this stage, exploration of application domains where P2P can better exploit its technological potential becomes vital. According to the Gartner report, it will take from 5 to 10 years before the real-world benefits of the P2P technology are demonstrated and accepted.

On the other hand, the emerging Semantic Web (SW) technologies open new horizons for P2P. The vision of the SW is to enable machines to retrieve and process meta-data (i.e., information about data), and exploit this knowledge for intelligent, meaningful and context-driven interoperation with users and other applications . In the SW P2P scenario, users encode their knowledge in a formal structure, such as ontology, and then share it with other users and applications; they create communities of knowing which gradually evolve as new knowledge is brought in. Moreover, the SW allows it to overcome current limitations of P2P data sharing applications. Namely, it allows for richer and mutually heterogeneous peers' schemas, while still ensuring interoperability. Note that when peers' schemas are heterogeneous, the role of centralized resource indexing servers diminishes, as there is no common schema to index resources.

Another knowledge-intensive domain is (organizational) Knowledge Management (KM). There is very little in the literature about what P2P can do for KM. However, Ovum reports that "Of all the application domains we have studied, knowledge management is the one where the benefits of peer-to-peer and a clear and straightforward business model for suppliers are most evident" [Axton 02]. Ovum identifies at least two major areas where P2P can have impact on KM: collaboration and knowledge discovery. The former area includes the support for virtual teams and organizations, and for communities of practice. The latter one is about finding content, social patterns and specialists' profiles in enterprise and personal P2P networks. At the technical level, these tasks require expressive formalisms to represent and share knowledge. From this point of view, SW technologies become very useful.

Currently, none of the commercialized P2P applications can be seen as a comprehensive KM solution. However lots of research is being done on this topic in academia. For instance, the Semantic Web community discusses the role of ontologies in KM systems (e.g., [Ehrig 03], [Fensel 02]). The database community proposes several solutions for a completely decentralized P2P database system with the support of heterogeneous schemas [Bernstein 02], [Halevy 03]. There are many other areas that contribute to building viable P2P KM solutions - personal knowledge management [Tsui 02], semantic matching of ontological structures to facilitate peers interoperability [see Giunchiglia 03], and so on.

3.2 Towards Distributed Knowledge Management

The traditional architecture of KM systems have embodied the assumption that, to share and exploit knowledge, it is necessary to implement a process of knowledge-extraction-and-refinement, whose aim is to eliminate all subjective and contextual aspects of knowledge, and create an objective and general representation that can then be reused by other people in a variety of situations [Bonifacio 00], [Bonifacio 02b], [Bonifacio 02a]. Very often, this process is finalized to build a central knowledge base, where knowledge can be accessed via a knowledge portal. This centralized approach - and its underlying objectivist epistemology - is one of the reasons why so many KM systems are deserted by users, who perceive such systems either as irrelevant or oppressive (see [Alvesson 01], [Tsoukas 01]). As clearly pointed by the Report on Knowledge Management to the European Commission, 2004 ”KM is a crucial competence in the new competitive arena but the degree of predictability which has been inherent in KM thinking, reflecting the general belief in linearity, is now seriously questioned.”

During the last year, the evidence that knowledge is a distributed, contextual and subjective matter have led to an alternative vision, the so called Distributed Knowledge Management (DKM). As described in [Bonifacio 02b], DKM is an approach to KM based on the principle that the multiplicity (and heterogeneity) of perspectives within complex organizations should not be viewed as an obstacle to knowledge exploitation, but rather as an opportunity that can foster innovation and creativity.

The fact that different individuals and communities may have very different perspectives, and that these perspectives affect their representation of the world (and therefore of their work) is widely discussed - and generally accepted - in theoretical research on the nature of knowledge. Knowledge representation in artificial intelligence and cognitive science has produced many theoretical and experimental evidences of the fact that what people know is not a mere collection of facts; indeed, knowledge always presupposes some (typically implicit) interpretation schema, which provides an essential component in sense-making (see, for example, the notions of context [McCarthy 93], [Bouquet 98], [Ghidini 01], mental space [Fauconnier 85], partitioned representation [Dinsmore 91]). Moreover, studies on the social nature of knowledge stress the social nature of interpretation schemas, viewed as the outcome of a special kind of ”agreement” within a community of knowing (see, for example, the notions of scientific paradigm [Kuhn 79], frame [Weick 95]), thought world [Dougherty 92], perspective [Boland 95]). In this sense, rather than linear, knowledge dynamics are better represented by evolving constellations of autonomous and heterogeneous ”knowledges” that consolidate their knowledge locally and seek for some form of coordination with other ”knowledges” by means of semantic negotiation.

Despite this large convergence, the need to preserve an idea of control which is inherent to the very notion of business resource, led organizations and managers to neglect what increasingly happens behind the scene of any corporate intranet: people and groups still continue to develop an interrelated web of local systems that better fit their needs. On the other hand, the risk that these constellations become unmanageable, and the maturity that is being reached by P2P technologies are increasingly attracting managers towards alternative perspectives. Among these, at least the opportunity to weakly control knowledge constellations making them visible, instead of hiding them behind the official but ephemeral claims of control provided by the knowledge portal. In this less ideal but more realistic landscape, P2P systems seem particularly suitable to implement this view in which the existence of knowledge ”lobbies” is recognized and inter-community cooperation is supported through the provision of adequate coordination facilities. In the next section, we propose a system that tries exactly to go in this direction.

3.3 Towards Distributed Knowledge Management

4 Peer-to-Peer Knowledge Management - Prospects and an Application The convergence of P2P and KM technologies creates new challenges for researchers to address: new methodologies to model, design, and deploy distributed KM solutions; theories and algorithms to represent the social and semantic dimensions of a knowledge network; mechanisms to cope with the dynamic autonomous nature of P2P and to provide means to support emergent network self-organization. New technologies should be provided in order to support full operational functioning of P2P KM systems, ensuring high extensibility of the solutions along several dimensions, such as scalability in the number of peers, size and kind of supported knowledge bases, level of heterogeneity in knowledge representation, etc. In return, P2P KM applications give the prospects of robustness, large pool of shared resources and semantics-driven tools to effectively operate these resources; local autonomy and comprehensive support to each peer provided by a collaborative effort of other peers. P2P KM solutions are not dependent on the presence of certain peers or content on the network; instead, peers bring new knowledge which flows along semantic links between peers, being enriched and completed on the fly. The ”knowledge base” of a P2P KM network is formed dynamically; peers forge and break knowledge groups based on a common interest, etc. From this point of view, it turns out that operation on a P2P KM network naturally complements usual economic and social patterns. As an example of an existing P2P KM system, we propose in this paper a P2P DKM architecture, named KEEx, which is the result of the research project EDAMOK (Enabling Distributed and Autonomous Management of Knowledge). In KEEx, each community of knowing (or Knowledge Nodes (KN), as they are called in [Bonifacio 02a]) is represented by a peer, and the two principles above are implemented in a quite straightforward way: (i) each peer provides all the services needed by a knowledge node to create and organize its own local knowledge (autonomy), and (ii) by defining social structures and protocols of meaning negotiation in order to achieve semantic coordination (e.g., when searching documents from other peers). Built on top of this architecture, distributed local ”knowledges” can emerge and aggregate through a bottom-up process from the individual level, to the organizational one, passing through the establishment of communities (group of peers that share a similar interest) and zones (networks of peers that relate to a neighbourhood). Moreover, peers' knowledge bases can be run on top of industrial solutions such as existing content management applications or databases. As a consequence, knowledge bases become to be virtual, flexible and temporary aggregation of both individual and more institutionalized knowledge sources.

Fig 1. The KEEx's main components The main components of KEEx are shown on Figure 1. Each Knowledge Node (also known as K-peers) can play two main roles:

provider and seeker. In the former case, a K-peer ”publishes” in the system a body of knowledge, together with an explicit representation on it. In the latter case, a K-peer searches for information by explicitly specifying a query as a part of its own perspective. K-peers store their local knowledge in document repositories. K-peers formally describe the real world (from their own perspective) in the approximate and partial form of a context. Contexts are also used by seekers for query representation. A K-peer may have more than one context, and it stores its context(s) in a context repository. Contexts are created, manipulated and used by K-peers by means of the context management module, which includes a context editor and a context browser. Apart from this, KEEx allows for semantic matching of contexts, for forming federations of K-peers based on knowledge that the peers have in common, and for peers discovery. KEEx is implemented on top of the P2P platform JXTA . For a throughout discussion of KEEx's components and functionality, see [Bonifacio 02c].

The Tropos methodology is intended to support all the analysis and design activities in the software development process, from the application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, that is incrementally refined and extended, providing a common interface to the various software development activities, as well as a basis for documentation and evolution of the software.

In the following, we introduce the five main development phases of the Tropos methodology: *Early Requirements*, *Late Requirements*, *Architectural Design*, *Detailed Design* and *Implementation*. We then define the basic notions to be modeled along these phases and the reasoning techniques that guide the model evolution. Finally, we describe the modeling activities performed along the five phases pointing out how the focus shifts following the process.

3.3.1 Development phases

Requirement analysis represents the initial phase in many software engineering methodologies. Similarly to other software engineering approaches, in Tropos the final goal of requirement analysis is to provide a set of functional and non-functional requirements for the system-to-be.

The requirements analysis in Tropos is split in two main phases: *Early Requirements* and *Late Requirements* analysis. Both share the same conceptual and methodological approach. Thus most of the ideas introduced for early requirements analysis are used for late requirements as well. More precisely, during the first phase, the requirements engineer identifies the domain stakeholders and models them as social actors, who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. By clearly defining these dependencies, it is then possible to state the *why*, beside the *what* and *how*, of the system functionalities and, as a last result, to verify how the final implementation matches the real needs. In the *Late Requirements* analysis, the conceptual model is extended including a new actor, which represents the system, and a number of dependencies with other actors part of the environment. These dependencies define all the functional and non-functional requirements of the system-to-be.

The *Architectural Design* and the *Detailed Design* focus on the system specification, according to the requirements resulting from the above phases. *Architectural Design* defines the system's global architecture in terms of subsystems, interconnected through data and control flows. Subsystems are represented, in the model, as actors and data/control interconnections are represented as dependencies. The architectural design provides also a mapping of the system actors to a set of software agents, each characterized by its specific capabilities. The *Detailed Design* phase aims at specifying the agent capabilities and interactions. At this point, usually, the implementation platform has already been chosen and this can be taken into account in order to perform a detailed design that will map directly to the code.[1]

The *Implementation* activity follows step by step, in a natural way, the detailed design specification on the basis of the established mapping between the implementation platform constructs and the detailed design notions.

3.3.2 The key concepts

Models in Tropos are acquired as instances of a *conceptual metamodel* resting on the following concepts/relationships:

Actor, which models an entity that has strategic goals and intentionality within the system or the organizational setting. An actor represents a physical or a software *agent* as well as a *role* or *position*. While we assume the classical AI definition of software agent, that is, a software having properties such as autonomy, social ability, reactivity, proactivity, as given, for instance in [Nwa96], in Tropos we define a *role* as an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor, and a *position* represents a set of roles, typically played by one agent. An agent can occupy a position, while a position is said to cover a role. A discussion on this issue can be found in [Yu01].

Goal, which represents actors' strategic interests. We distinguish hard goals from softgoals, the second having no clear-cut definition and/or criteria for deciding whether they are satisfied or not. According to [CNYM00], this different nature of achievement is underlined by saying that goals are *satisfied* while softgoals are *satisficed*. Softgoals are typically used to model non-functional requirements.

Plan, which represents, at an abstract level, a way of doing something. The execution of plan can be a means for satisfying a goal or for satisficing a softgoal.

[1]Notice that Tropos (and we believe also the other agent-oriented software engineering methodologies) can be used independently of the fact that one uses AOP as implementation technology.

Resource, which represents a physical or an informational entity.

Dependency between two actors, which indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource. The former actor is called the *depender*, while the latter is called the *dependee*. The object around which the dependency centers is called *dependum*. In general, by depending on another actor for a dependum, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Capability, which represents the ability of an actor of defining, choosing and executing a plan for the fulfillment of a goal, given certain world conditions and in presence of a specific event.

Belief, which represents the actor knowledge of the world.

These notions are more formally specified in the language metamodel described in Section 3.6.

3.3.3 Modeling activities

Various activities contribute to the acquisition of a first early requirement model, to its refinement and to its evolution into subsequent models. They are:

Actor modeling, which consists of identifying and analyzing both the actors of the environment and the system's actors and agents. In particular, in the early requirement phase actor modeling focuses on modeling the application domain stakeholders and their intentions as social actors which want to achieve goals. During late requirement, actor modeling focuses on the definition of the system-to-be actor, whereas in architectural design, it focuses on the structure of the system-to-be actor specifying it in terms of subsystems (actors), interconnected through data and control flows. In detailed design, the system's agents are defined specifying all the notions required by the target implementation platform, and finally, during the implementation phase actor modeling corresponds to the agent coding.

Dependency modeling, which consists of identifying actors which depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. In particular, in the early requirement phase, it focuses on modeling goal dependencies between social actors of the organizational setting. New dependencies are elicited and added to the model upon goal analysis performed during the goal modeling activity discussed below. During late requirements analysis, dependency modeling focuses on analyzing the dependencies of the system-to-be actor.

In the architectural design phase, data and control flows between sub-actors of the system-to-be actors are modeled in terms of dependencies, providing the basis for the capability modeling that will start later in architectural design together with the mapping of system actors to agents.

A graphical representation of the model obtained following these modeling activities is given through *actor diagrams* (see Section 3.6 for more details), which describe the actors (depicted as circles), their goals (depicted as ovals and cloud shapes) and the network of dependency relationships among actors (two arrowed lines connected by a graphical symbol varying according to the dependum: a goal, a plan or a resource). An example is given in Figure 3.1.

Goal modeling rests on the analysis of an actor goals, conducted from the point of view of the actor, by using three basic reasoning techniques: *means-end analysis*, *contribution analysis*, and *AND/OR decomposition*. In particular, means-end analysis aims at identifying plans, resources and softgoals that provide means for achieving a goal. Contribution analysis identifies goals that can contribute positively or negatively in the fulfillment of the goal to be analyzed. In a sense, it can be considered as an extension of means-end analysis, with goals as means. AND/OR decomposition combines AND and OR decompositions of a root goal into sub-goals, modeling a finer goal structure. Goal modeling is applied to early and late requirement models in order to refine them and to elicit new dependencies. During architectural design, it contributes to motivate the first decomposition of the system-to-be actors into a set of sub-actors.

Plan modeling can be considered as an analysis technique complementary to goal modeling. It rests on reasoning techniques analogous to those used in goal modeling, namely, means-end, contribution analysis and AND/OR decomposition. In particular, AND/OR decomposition provides an AND and OR decompositions of a root plan into sub-plans.

A graphical representation of goal and plan modeling is given through *goal diagrams*, see, for instance, Figure 3.3 but also Section 3.6 for more details.

Capability modeling starts at the end of the architectural design when system sub-actors have been specified in terms of their own goals and the dependencies with other actors. In order to define, choose and execute a plan for achieving its own goals, each system's sub-actor has to be provided with specific "individual" capabilities. Additional "social" capabilities should be also provided for managing dependencies with other actors. Goals and plans previously modeled become integral part of the capabilities. In detailed design, each agent's capability is further specified and then coded during the implementation phase.

A graphical representation of these capabilities is given by *capability* and *plan diagrams*. UML activity diagrams (see Figure 4.6 for an example) and AUML interaction diagrams [OPB00] (Figure 4.8) are used to this purpose (more details in Section 3.6).

3.4 An example

In this section we go through and discuss the five Tropos phases via a substantial case study. The example considered is a fragment of a real application developed for the government of Trentino (Provincia Autonoma di Trento, or PAT). In the exposition, the example has been suitably modified to take into account a non disclosure agreement and also to make it simpler and therefore more easily understandable. The system (which we will call throughout the *eCulture system*) is a web-based broker of cultural information and services for PAT, including information obtained from museums, exhibitions, and other cultural organizations and events [GPM+01b]. It is the government's intention that the system be usable by a variety of users, including Trentino citizens and tourists, looking for things to do, or scholars and students looking for material relevant to their studies.

3.4.1 Early Requirements Analysis

Early Requirements analysis consists of identifying and analyzing the stakeholders and their intentions. Stakeholders are modeled as social actors who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. Intentions are modeled as goals which, through a goal-oriented analysis, are decomposed into finer goals, that eventually can support evaluation of alternatives.

In our eCulture example we can start by informally listing (some of) the stakeholders:

- Provincia Autonoma di Trento (PAT), that is the government agency funding the project; its objectives include improving public information services, increasing tourism through new information services, also encouraging Internet use within the province.
- Museums, that are the major cultural information providers for their respective collections; museums want government funds to build/ improve their cultural information services, and are willing to interface their systems with other cultural systems or services.
- Visitors, who want to access cultural information, before or during their visit to Trentino, to make their visit interesting and/or pleasant.
- (Trentino) Citizens, who want easily accessible information, of any sort, and (of course) good administration of public resources.

Figure 3.1 shows the *actor diagram* for the eCulture domain. In particular, Citizen is associated with a single relevant goal: get cultural information, while Visitor has an associated softgoal enjoy visit. Along similar lines, PAT wants to increase internet use while Museum wants to provide cultural services. Finally, the diagram includes one softgoal dependency where Citizen depends on PAT to fulfill the taxes well spent softgoal.

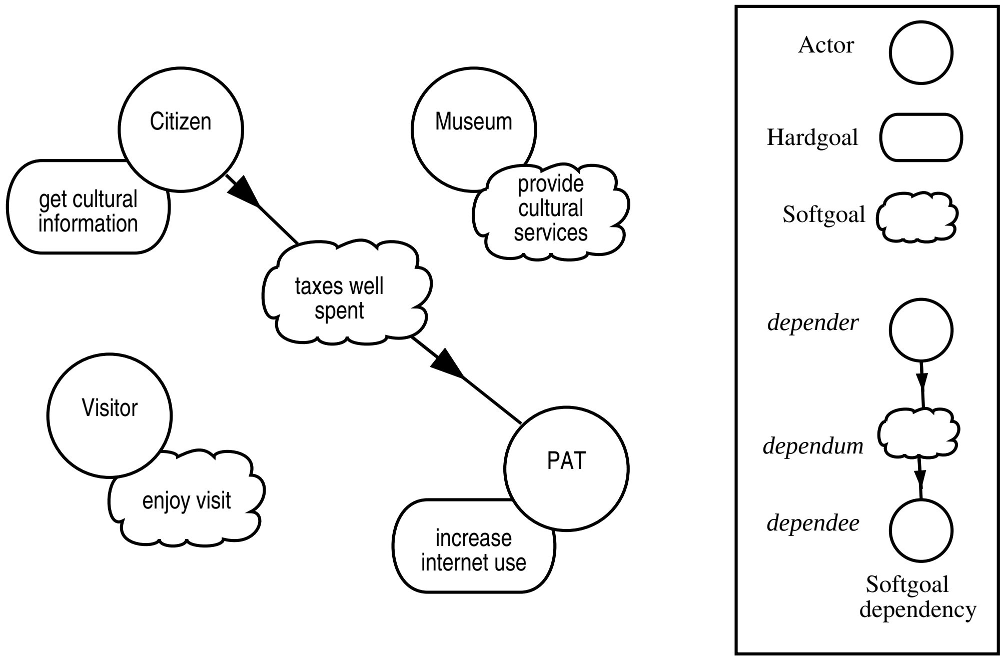


Figure 3.1: Actor diagram modeling the stakeholders of the eCultural project.

Once the stakeholders have been identified, along with their goals and social dependencies, the analysis proceeds in order to enrich the model with further details. In particular, the rationale of each goal relatively to the stakeholder who is responsible for its fulfillment has to be analyzed. Basically, this is done through means-end analysis and goal/plan decomposition.

A first example of the result of such an analysis from the perspective of Citizen and Visitor is given by the *goal diagrams* depicted in Figure 3.2. For the actor Citizen, the goal get cultural information is decomposed into visit cultural institutions and visit cultural web systems. These two subgoals can be seen as alternative ways of fulfilling the goal get cultural information (and we will call this a “OR-decomposition”). Goal decomposition can be closed through a means-end analysis aimed at identifying plans, resources and softgoals that provide means for achieving the goal. For example, the plan (depicted as a hexagon) visit eCulture System is a means to fulfill the goal visit cultural web systems. This plan can be decomposed into sub-plans, namely use eCulture System and access internet. These two sub-plans become the reasons for a set of dependencies between Citizen and PAT: eCulture System available, internet infrastructure available and usable eCulture System. The analysis for Visitor is simpler: planning a visit can give a positive contribution to the goal enjoy visit, and for this the Visitor needs the eCulture System too.

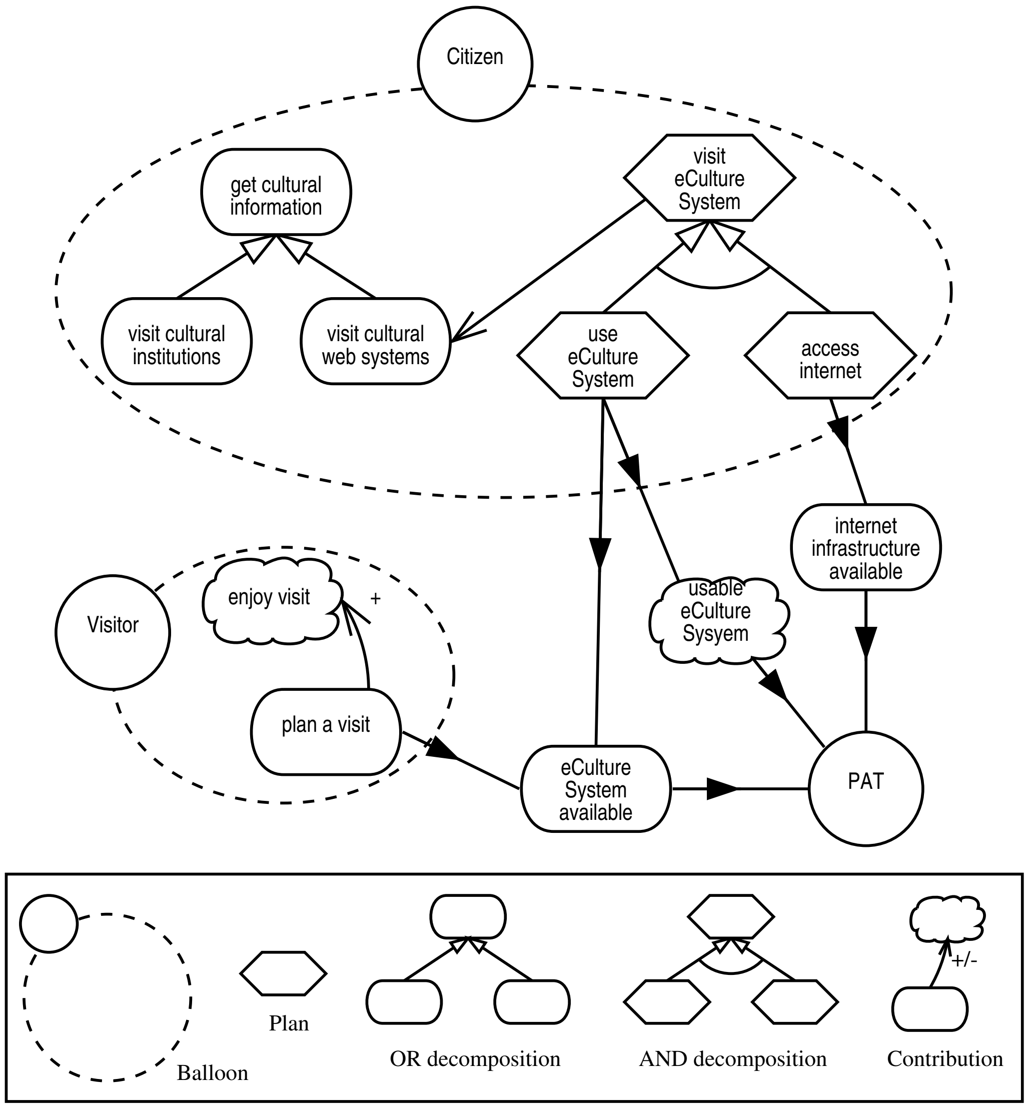


Figure 3.2: Goal diagrams for Citizen and Visitor. Notice the goal and plan decomposition, the means-end analysis and the (positive)softgoal contribution.

A second example, in Figure 3.3, shows portions of the goal analysis for PAT, relatively to the goals that Citizen delegates to PAT as a result of the previous analysis. The goals increase internet use and eCulture System available are both well served by the goal build eCulture System. Inside the *actor diagram*, softgoal analysis is performed identifying the goals that contribute positively or negatively to the softgoal. The softgoal taxes well spent gets positive contributions from the softgoal good services, and, in the end, from the goal build eCulture System too.

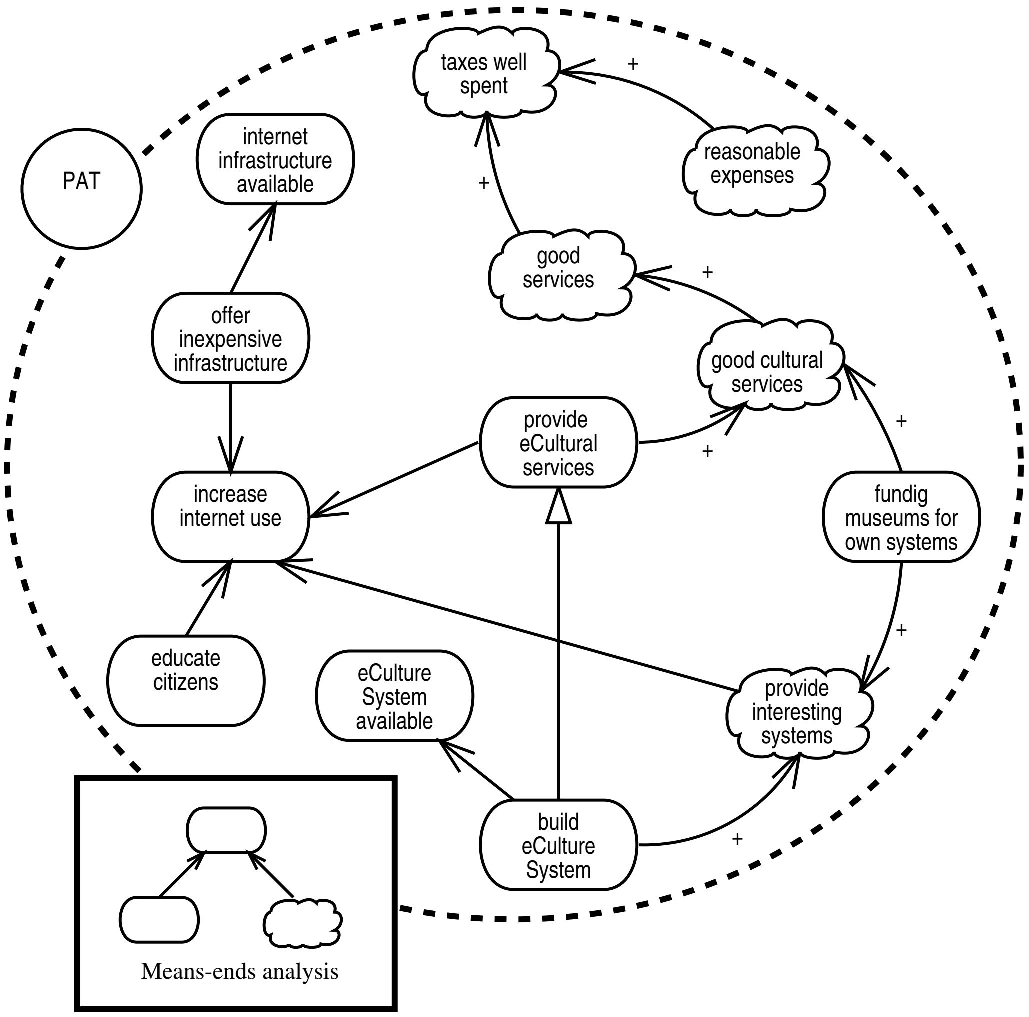

taxes well spent
+
reasonable expenses
+
PAT
internet infrastructure available
good services
+
offer inexpensive infrastructure
good cultural services
provide eCultural services
+
+
increase internet use
fundig museums for own systems
+
educate citizens
eCulture System available
provide interesting systems
build eCulture System
+
Means-ends analysis

Figure 3.3: Goal diagram for PAT.

The final result of this phase is a set of strategic dependencies among actors, built incrementally by performing goal/plan analysis on each goal, until all goals have been analyzed. The later it is added, the more specific a goal is. For instance, in the example in Figure 3.3, the goal build eCulture System is introduced last and, therefore, it has no sub-goals and it is motivated by the higher level goals.

3.4.2 Late Requirements Analysis

Late requirement analysis focuses on the system-to-be (the eCulture System in our case) within its operating environment, along with relevant functions and qualities. The system-to-be is represented as one actor which has a number of dependencies with the other actors of the organization. These dependencies define the system's functional and non-functional requirements.

The actor diagram in Figure 3.4 includes the eCulture System and shows a set of goals and softgoals that PAT delegates to it. In particular, the goal provide eCultural services, which contributes to the main goal of PAT increase internet use (see Figure 3.3), and the softgoals extensible eCulture System, flexible eCulture System, usable eCulture System, and use internet technology. These goals are then analyzed from the point of view of the eCulture System. In Figure 3.4 we concentrate on the analysis of the goal provide eCultural services and the softgoal usable eCulture System. The goal provide eCultural services is decomposed (AND decomposition) into four subgoals: make reservations, provide info, educational services and virtual visits. As basic eCultural service, the eCulture System must provide information (provide info), which can be logistic info, and cultural info. Logistic info concerns, for instance, timetables and visiting instructions for museums, while cultural info concerns the cultural content of museums and special cultural events. This content may include descriptions and images of historical objects, the description of an exhibition, and the history of a particular region. Virtual visits are services that allow, for instance, Citizen to pay a virtual visit to a city of the past (Rome during Cæsar's time!). Educational services includes presentation of historical and cultural material at different levels (e.g., high school or undergraduate university level) as well as on-line evaluation of the student's grasp of this material. Make reservations allows the Citizen to make reservations for particular cultural events, such as concerts, exhibitions, and guided museum visits.

Softgoal contributions can be identified applying the same kind of analysis described by the goal diagram of Figure 3.3. So for instance, the softgoal usable eCulture System has two positive (+) contributions from softgoals user friendly eCulture System and available eCulture System. The former contributes positively because a system must be user friendly to be usable, whereas the latter contributes positively because it makes the system portable, scalable, and available over time (temporal available).

Often, some dependencies in the actor diagram must be revised upon the introduction of the system actor. We have seen in Figure 3.2 that for Citizen a possible subplan of getting eCultural info is using an eCulture system. Now we can model this in terms of a direct dependency between the actors Citizen and eCulture System. Figure 3.5 shows how this dependency is analyzed inside the goal diagram of the eCulture System. The goal search information (a subgoal of the goal provide info) can be fulfilled by four different plans: search by area (thematic area), search by geographical area, search by keyword, and search by time period. The decomposition into sub-plans is almost the same for all four kinds of search. For example, the sub-plan get info on area is decomposed in find info sources, that finds which information sources are more appropriate to provide information concerning the specified area, and the sub-plan query sources, that queries the information sources. The sub-plan find info sources depends on the museums for the description of the information that the museums can provide, i.e., the resource dependency info about source (a rectangle in Figure 3.5), and synthesize results depends on museums for query result. Finally, in order to search information about a particular thematic area, the Citizen is required to provide information using an area specification form.

3.4.3 Architectural Design

The architectural design phase defines the system's global architecture in terms of subsystems (actors) interconnected through data and control flows (dependencies). This phase is articulated in three steps, as follows.

Step 1. As first step, the overall architectural organization is defined. New actors (including sub-actors) are introduced in the system as a result of analysis performed at different levels of abstraction, such as:

- inclusion of new actors and delegation of subgoals to sub-actors upon goal analysis of system's goals;
- inclusion of new actors according to the choice of a specific architectural style (see [FGKM01, KGM01] for more details about the use of architectural patterns and styles);
- inclusion of actors contributing positively to the fulfillment of specific functional and non-functional requirements.

Figure 3.6 shows the decomposition in sub-actors of the eCulture System and the delegation of some goals from the eCulture System to them. The eCulture System depends on the Info Broker to provide info, on the Educational Broker to provide educational services, on the Reservation Broker to make reservations, on Virtual Visit Broker to provide virtual visits, and on System Manager to provide interface. Additionally, each sub-actor can be itself decomposed in sub-actors responsible for the fulfillment of one or more sub-goals.

The final result of this first step is an extended actor diagram, in which new actors and their dependencies with the other actors are presented. Figure 3.7 shows the extended actor diagram with respect to the Info Broker and the assigned plan search by area. The User Interface Manager and the Sources Interface Manager are responsible for interfacing the system to the external actors Citizen and Museum. The Services Broker and Sources Broker have been also introduced to facilitate generic interactions outside the system. Services Broker manages a repository of descriptions for services offered by actors within the eCulture System. Analogously, Sources Broker manages a repository of descriptions for information sources available outside the system.

The three sub-actors: the Area Classifier, the Results Synthesizer, and the Info Searcher (Figure 3.7) have been introduced upon the analysis of the plan search by area reported in Figure 3.5. Area Classifier is responsible for the classification of the information provided by the user. It depends on the User Interface Manager for interfacing to the users, and on the Service Broker to have information about the services provided by other actors. The Info Searcher depends on Area Classifier to have information about the thematic area that the user is interested in, on the Source Broker for the description of the information sources available outside the system, and on the Sources Interface Manager for interfacing to the sources. The Results Synthesizer depends on the Info Searcher for the information concerning the query that the Info Searcher asked, and on the Museum to have the query results.

Actor Name	**N**	**Capability**
Area Classifier	1	get area specification form
	2	classify area
	3	provide area information
	4	provide service description
Info Searcher	5	get area information
	6	find information source
	7	compose query
	8	query source
	9	provide query information
		provide service description
Results Synthesizer	10	get query information
	11	get query results
	12	provide query results
	13	synthesize area query results
		provide service description
Sources Interface Manager	14	wrap information source
		provide service description
Sources Broker	15	get source description
	16	classify source
	17	store source description
	18	delete source description
	19	provide sources information
		provide service description
Services Broker	20	get service description
	21	classify service
	22	store service description
	23	delete service description
	24	provide services information
User Interface Manager	25	get user specification
	26	provide user specification
	27	get query results
	28	present query results to the user
		provide service description

Table 3.1: Actors' capabilities (step 2).

Step 2. This step consists in the identification of the capabilities needed by the actors to fulfill their goals and plans. Capabilities can be easily identified by analyzing the extended actor diagram. In particular, each dependency relationship can give place to one or more capability triggered by external events. To give an intuitive idea of this process let's focus on a specific actor of the extended actor diagram, such as the Area Classifier, and consider all the in-going and out-going dependencies, as shown in Figure 3.8. Each dependency is mapped to a capability. So, for instance, the dependency for the resource area specification form calls for the capability get area specification form, and so on. The Area Classifier's capabilities as well as the capabilities of the other actors of the extended actor diagram of Figure 3.7 are listed in Table 3.1.

Step 3. The last step consists of defining a set of agent types and assigning each of them one or more different capabilities (agent assignment). Table 3.2 reports the agents assignment with respect to the capabilities identified in Table 3.1. Of course, many other capabilities and agent types are needed in case we consider all the goals and plans associated to the complete extended actor diagram.

Agent	**Capabilities**
Query Handler	1, 3, 4, 5, 7, 8, 9, 10, 11, 12
Classifier	2, 4
Searcher	6, 4
Synthesizer	13, 4
Wrapper	14, 4
Agent Resource Broker	15, 16, 17, 18, 19, 4
Directory Facilitator	20, 21, 22, 23, 24, 4
User Interface Agent	25, 26, 27, 28, 4

Table 3.2: Agent types and their capabilities.

In general, the agents assignment is not unique and depends on the designer. The number of agents and the capabilities assigned to each of them are choices driven by the analysis of the extend actor diagram and by the way in which the designer think the system in term of agents. Tropos offers a set of pre-defined patterns recurrent in multi-agent literature that can help the designer [KGM01].

3.5 The Development Process

The previous sections introduced the primitive concepts supported by Tropos and the different kinds of modeling activities one performs during a Tropos-based software development project. In this section, we focus on the generic design process through which these models are constructed. The process is basically one of analyzing goals on behalf of different actors, and is described in terms of a non deterministic concurrent algorithm, including a completeness criterion. Note that this process is carried out by *software engineers* (rather than software agents) at *design-time* (rather than run-time). We present in this section only the algorithms for early and late requirements analysis.

Intuitively, the process begins with a number of actors, each with a list of associated root goals (possibly including softgoals). Each root goal is analyzed from the perspective of its respective actor, and as subgoals are generated, they are delegated to other actors, or the actor takes on the responsibility of dealing with them him/her/itself. This analysis is carried out concurrently with respect to each root goal. Sometimes the process requires the introduction of new actors which are delegated goals and/or tasks. The process is complete when all goals have been dealt with to the satisfaction of the actors who want them (or the designers thereof.)

Assume that `actorList` includes a finite set of actors, also that the list of goals for `actor` is stored in `goalList(actor)`. In addition, we assume that `agenda(actor)` includes the list of goals `actor` has undertaken to achieve personally (with no help from other actors), along with the plan that has been selected for each goal. Initially, `agenda(actor)` is empty. `dependencyList` includes a list of dependencies among actors, while `capabilityList(actor)` includes $\langle \texttt{goal}, \texttt{plan} \rangle$ pairs indicating the means by which the actor can achieve particular goals. Finally, `goalGraph` stores a representation of the goal graph that has been generated so far by the design process. Initially, `goalGraph` contains all root goals of all initial actors with no links among them. We will treat all of the above as global variables which are accessed and/or updated by the procedures presented below. For each procedure, we use as parameters those variables used within the procedure.

```
global actorList, goalList, agenda, dependencyList,
      capabilityList, goalGraph;
procedure rootGoalAnalysis(actorList, goalList, goalGraph)
   begin
     rootGoalList = nil;
     for actor in actorList do
       for rootGoal in goalList(actor) do
         rootGoalList = add(rootGoal, rootGoalList);
         rootGoal.actor = actor;
       end ;
     end ;
   end ;
   concurrent for rootGoal in rootGoalList do
     goalAnalysis(rootGoal, actorList)
   end concurrent for ;
         if not[satisfied(rootGoalList, goalGraph)]
               then fail;
   end procedure
```

The procedure `rootGoalAnalysis` conducts concurrent goal analysis for every root goal. Initially, root goal analysis is conducted for all initial goals associated with actors in `actorList`. Later on, more root goals are created as goals are delegated to existing or new actors. Note that the **concurrent for** statement spawns a concurrent call to `goalAnalysis` for every element of the list `rootGoalList`. Moreover, more calls to `goalAnalysis` are spawn as more root goals are added to `rootGoalList`. **concurrent for** is assumed to terminates when all its threads do. The predicate `satisfied` checks whether all root goals in `goalGraph` are satisfied. This predicate is computed in terms of a label propagation algorithm such as the one described in [MCN92]. Its details are beyond the scope of this chapter. `rootGoalAnalysis` succeeds if there is a set of non-deterministic selections within the concurrent executions of `goalAnalysis` procedures which leads to the satisfaction of all root goals.

The procedure `goalAnalysis` conducts concurrent goal analysis for every subgoal of a given root goal. Initially, the root goal is placed in `pendingList`. Then, **concurrent for** selects concurrently goals from `pendingList` and for each decides non-deterministically whether it will be expanded, adopted as a personal goal, delegated to an existing or new actor, or whether the goal will be treated as unsatisfiable (`'denied'`). When a goal is expanded, more subgoals are added to `pendingList` and `goalGraph` is augmented to include the new goals and their relationships to their parent goal. Note that the selection of an actor to delegate a goal is also non-deterministic, and so is the creation of a new actor. The three non-deterministic operations in `goalAnalysis` are highlighted with italic-bold font. These are the points where the designers of the software system will use their creative in designing the system-to-be.

```
procedure goalAnalysis(rootGoal, actorList)
  pendingList = add(rootGoal, nil);
  concurrent for goal in pendingList do
    decision = decideGoal(goal)
    case of decision
     expand :
      begin
        newGoalList = expandGoal(goal, goalGraph);
        for newGoal in newGoalList do
          newGoal.actor = goal.actor;
          add(newGoal, pendingList);
        end ;
      end ;
     solve : acceptGoal(goal, agenda(goal.actor));
     delegate :
      begin
        actor = selectActor(actorList);
        delegateGoal(goal, actor, rootGoalList, dependencyList);
      end ;
      newActor :
        begin
          actor = newActor(goal);
          actorList = add(actor, actorList);
          delegateGoal(goal, actor, rootGoalList, dependencyList);
        end ;
      fail : goal.label ='denied';
    end case of ;
  end concurrent for ;
end procedure
```

Finally, we specify two of the sub-procedures used in `goalAnalysis`, for the lack of space, others are left to the imagination of the reader. `delegateGoal` adds a goal to an actor's goal list because that goal has been delegated to the actor. This goal now becomes a root goal (with respect to the actor it has been delegated to), so another call to `goalAnalysis` is spawn by `rootGoalAnalysis`. Also, `dependencyList` is updated. The procedure `acceptGoal` simply selects a plan for a goal the actor will handle personally from the actor's capability list. The process we present here does not provide for extensions to a capability list to deal with a newly assigned goal.

```
procedure delegateGoal(goal, toActor, rootGoalList,
      dependencyList)
  begin
    add(goal, goalList(toActor));
    add(goal, rootGoalList);
    goal.actor = toActor;
    add(⟨goal.actor, toActor, goal⟩, dependencyList);
  end
end procedure
```

```
procedure acceptGoal(goal, agenda)
  begin
    plan = selectPlan(goal, capabilityList(goal.actor));
    add(⟨goal, plan⟩, agenda(goal.actor));
    goal.label ='satisfied';
  end
end procedure
```

During early requirements, this process analyzes initially-identified goals of external actors ("stakeholders"). At some point (late requirements), the system-to-be is introduced as another actor and is delegated some of the subgoals that have been generated from this analysis. During architectural design, more system actors are introduced and are delegated subgoals to system-assigned goals. Apart from generating goals and actors in order to fulfill initially-specified goals of external stakeholders, the development process includes specification steps during each phase which consist of further specifying each node of a model such as those shown in Figures 3-4. Specifications are given in a formal language (*Formal Tropos*) described in detail in [FPMT01]. These specifications add constraints, invariants, pre- and post-conditions which capture more of the semantics of the subject domain. Moreover, such specifications can be simulated using model checking technology for validation purposes [FPMT01, CCGR00].

Level	**Description**	**Examples**
Meta-Metamodel	Specifies language structural elements	Attribute, Entity
Metamodel	An instance of the meta-metamodel Defines knowledge level notions	Actor, Goal, Plan
Domain	An instance of the metamodel Models application domain entities	PAT, Citizen, Museum
Instance	Instantiates domain model elements	John: instance of Citizen

Table 3.3: Tropos language metamodel. The four level architecture.

3.6 The modeling language

The modeling language is at the core of the Tropos methodology. In this section, the abstract syntax of the language is defined in terms of a UML metamodel. Following standard approaches [OMG99], Tropos has exploited standard four-layer metadata architecture, as shown in Table 3.3. The four-layer architecture makes the Tropos language extensible in the sense that new constructs can be added. The semantics of the language (augmented with a powerful fragment of Temporal Logic [CE81]) is handled in [FPMT01] and will not be discussed here.

The *Meta-Metamodel* level provides the basis for the metamodel language. In particular, the meta-metamodel contains language primitives that allows for the inclusions of constructs such as those proposed in [FPMT01]. The *Metamodel* level provides constructs for modeling knowledge level entities and concepts. The *Domain* level contains a representation of entities and concepts of a specific application domain, built as instances of the metamodel level constructs. So, for instance, the examples used in Section 3.3 illustrate portions of the eCulture domain model. The *Instance* level contains instances of the domain model.

Before moving to the details of the metamodels for the concepts actor, goal and plan[2], let us present the Tropos model and diagrams.

A Tropos model is a directed labeled graph whose nodes are instances of metaclasses of the metamodel, namely *actor*, *goal*, *plan* and *resource*, and whose arcs are instances of the metaclasses representing relationships between them, *dependency*, *means-end analysis*, *contribution* and *AND/OR decomposition*.

Each element in the model has its own graphical representation. In particular, we use two types of diagram for visualizing the model: the *actor diagram* and the *goal diagram*.

An *actor diagram* is a graph, where each node represents an actor, and each arc represents a dependency between the two connecting nodes. The arc is labeled by a specific dependum. Examples of simple actor diagrams have been presented in Figure 3.1 and in Figure 3.6.

A *goal diagram* represents the perspective of a specific actor. It is drawn as a balloon and contains graphs whose nodes are goals (ovals) and /or plans (hexagonal shape) and whose arcs are the different relationships that can be identified among its nodes.

UML activity diagrams and AUML interaction diagrams are used to represent, respectively, properties (*capability* and *plan diagrams*) and agents' interaction.

According to the specific process development phase we are considering, we can define different views of the model. For instance, the *early requirement* view of the model will be composed of a set of actor and goal diagrams concerning the social actors modeling, while the *detailed design* view will be composed of a set of AUML diagrams specifying the agents's microlevel.

3.6.1 The concept of Actor

A portion of the Tropos metamodel concerning the concept of actor is shown in the UML class diagram of Figure 3.9. *Actor* is represented as a UML class. An actor can have $0 \ldots n$ goals. The UML class *Goal* represents here both hard and softgoals. A goal is wanted by $0 \ldots n$ actors, as specified by the UML association relationship. An actor can have $0 \ldots n$ beliefs and, conversely, beliefs are believed by $1 \ldots n$ actors.

An actor *dependency* is a quaternary relationship represented as a UML class. A dependency relates respectively a depender, dependee, and dependum (as defined earlier), also an optional reason for the dependency (labelled *why*). Examples of dependency relationships are shown in Figures 3.1, 3.4, and 3.6. The early requirements model depicted in Figure 3.1, for instance, shows a softgoal dependency between the actors Citizen and PAT. Its dependum is the softgoal taxes well spent, while the actors Citizen and PAT play the roles of depender and dependee, respectively.

[2]The metamodels concerning the other concepts are defined analogously with the partial description reported here. A complete description of the Tropos language metamodel can be found in [SPG01].

Figure 3.2 includes some why associations, depicted as arrows coming out from the inside of actors' balloon (e.g., the Citizen) and pointing towards dependums. For example, the plan use eCulture System plays the role of why for both the dependums usable eCulture System (softgoal) and eCulture System available (goal).

3.6.2 The concept of Goal

The concept of goal is represented by the class *Goal* in the UML class diagram depicted in Figure 3.10. The distinction between hard and softgoals is captured through a specialization of *Goal* into subclasses *Hardgoal* and *Softgoal*, respectively.

Goals can be analyzed, from the point of view of an actor, performing *means-end analysis*, *contribution analysis* and *AND/OR decomposition* (listed in order of strength). Let us consider these in turn.

Means-end Analysis is a ternary relationship defined among an *actor*, whose point of view is represented in the analysis, a goal (the end), and a *Plan*, *Resource* or *Goal* (the means). Means-end analysis is used in the model shown in Figure 3.3, where the goals educate citizens and provide eCultural services, as well as the softgoal provide interesting systems are means for achieving the goal increase internet use.

Contribution Analysis is a ternary relationship between an *actor*, whose point of view is represented, and two goals. Contribution analysis strives to identify goals that can contribute positively or negatively towards the fulfillment of a goal (see association relationship labelled *contributes to* in Figure 3.10). A contribution can be annotated with a qualitative metric, as used in [CNYM00], denoted by $+, ++, -, --$. In particular, if the goal *g1* contributes positively to the goal *g2*, with metric $++$ then if *g1* is satisfied, so is *g2*. Analogously, if the plan *p* contributes positively to the goal *g*, with metric $++$, this says that *p* fulfills *g*. A $+$ label for a goal or plan contribution represents a partial, positive contribution to the goal being analyzed. With labels $--$, and $-$ we have the dual situation representing a sufficient or partial negative contribution towards the fulfillment of a goal. Examples of contribution analysis are shown in Figure 3.3. For instance, the goal funding museums for own systems contributes positively to both the softgoals provide interesting systems and good cultural services, and the latter softgoal contributes positively to the softgoal good services.

Contribution analysis applied to softgoals is often used to evaluate non-functional (quality) requirements.

AND/OR Decomposition is also a ternary relationship which defines an *AND-* or *OR-decomposition* of a root goal into subgoals. The particular case where the root goal *g1* is decomposed into a single subgoal *g2*, is equivalent to a $++$ contribution from *g2* to *g1*.

3.6.3 The concept of Plan

The concept of plan in Tropos is specified by the class diagram depicted in Figure 3.11. *Means-end analysis* and *AND/OR decomposition*, defined above for goals, can be applied to plans also. In particular, *AND/OR decomposition* allows for modeling the plan structure.

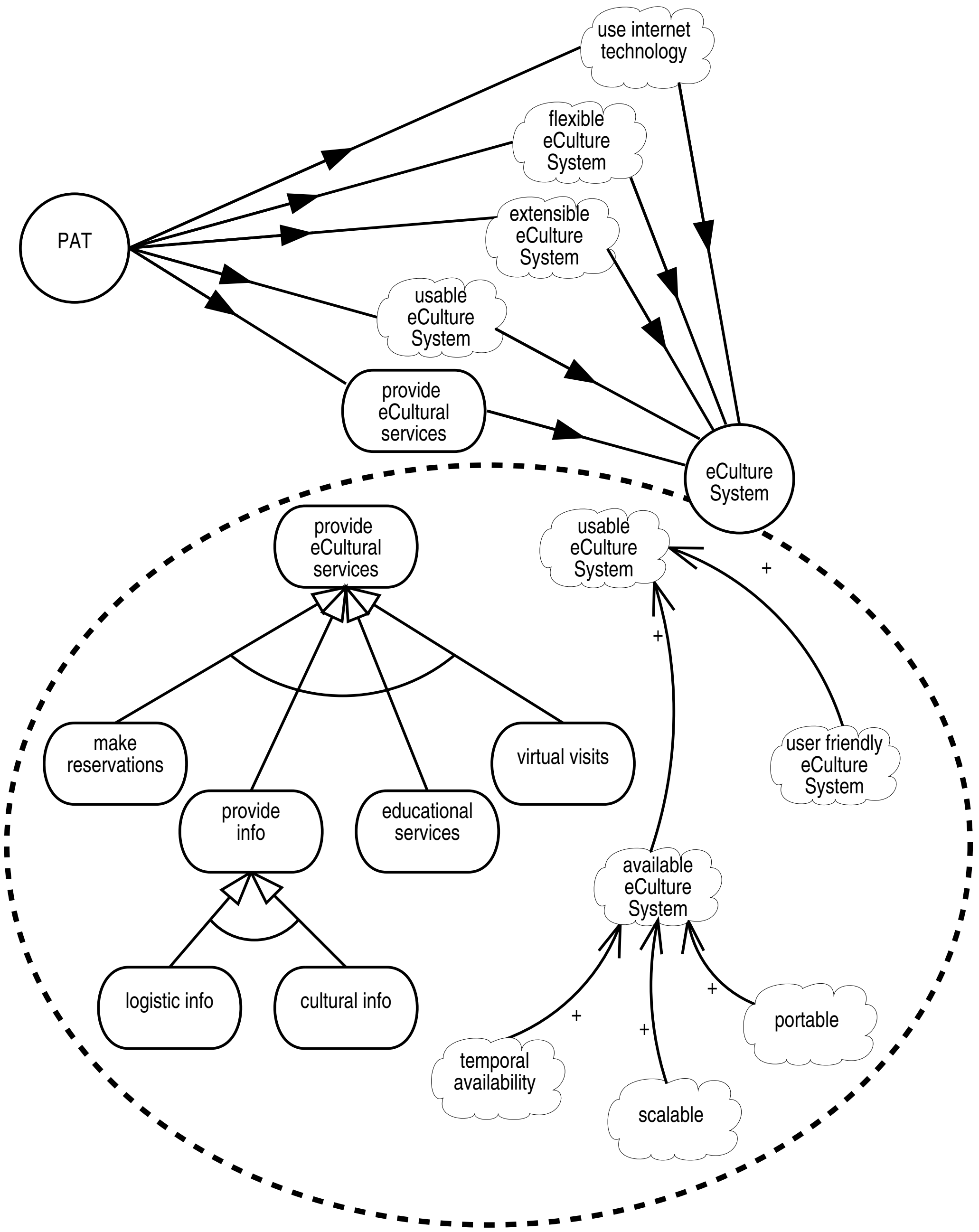


Figure 3.4: A portion of the actor diagram including PAT and eCulture System and goal diagram of the eCulture System.

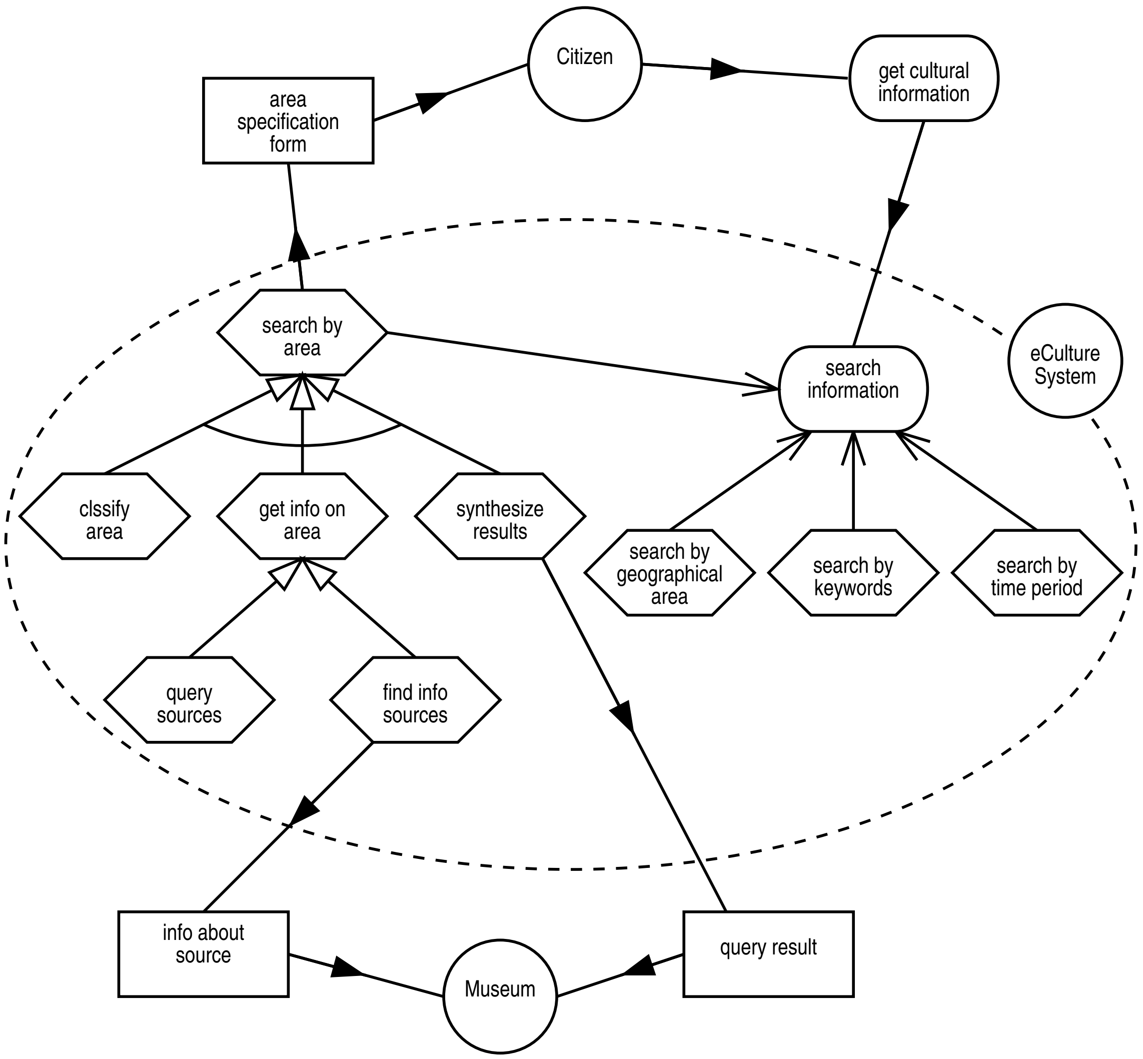


Figure 3.5: Goal diagram for the goal get cultural information and dependencies between the actor eCulture System and other environment' actors.

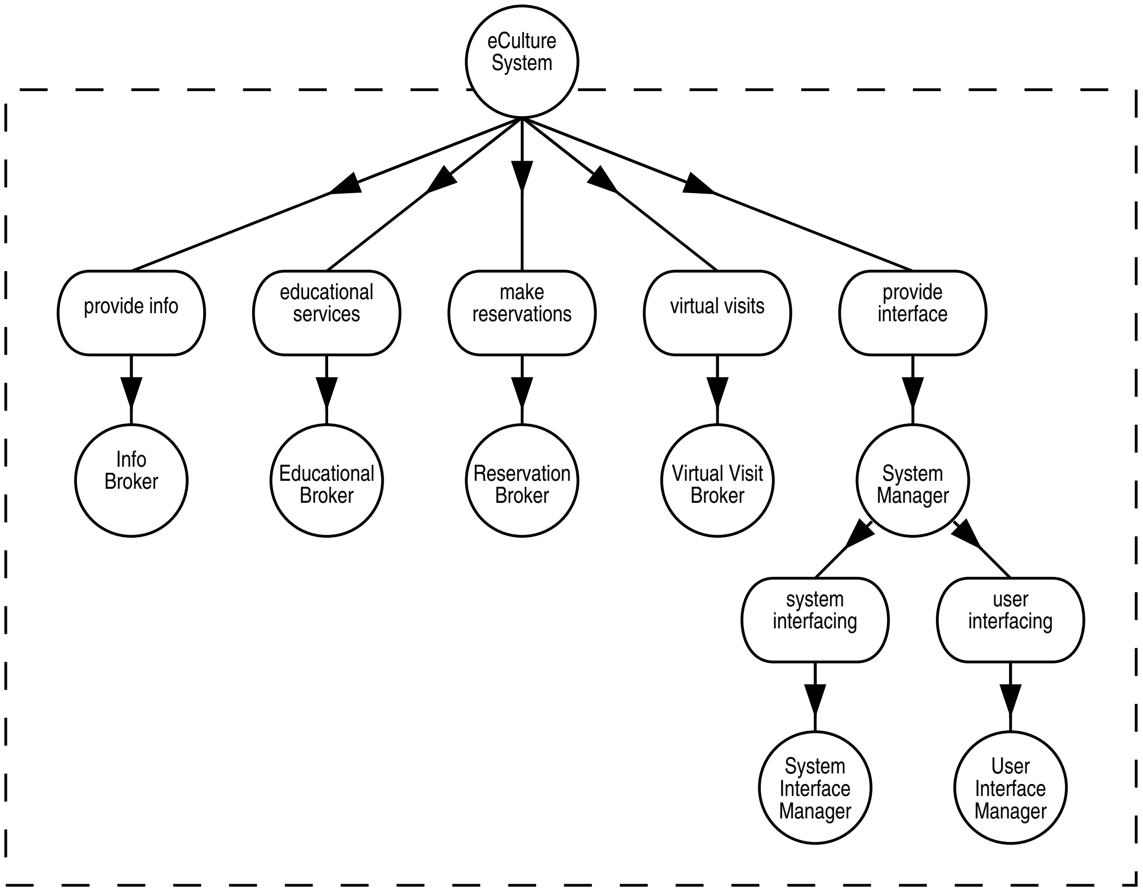


Figure 3.6: Actor diagram for the eCulture System architecture (step 1).

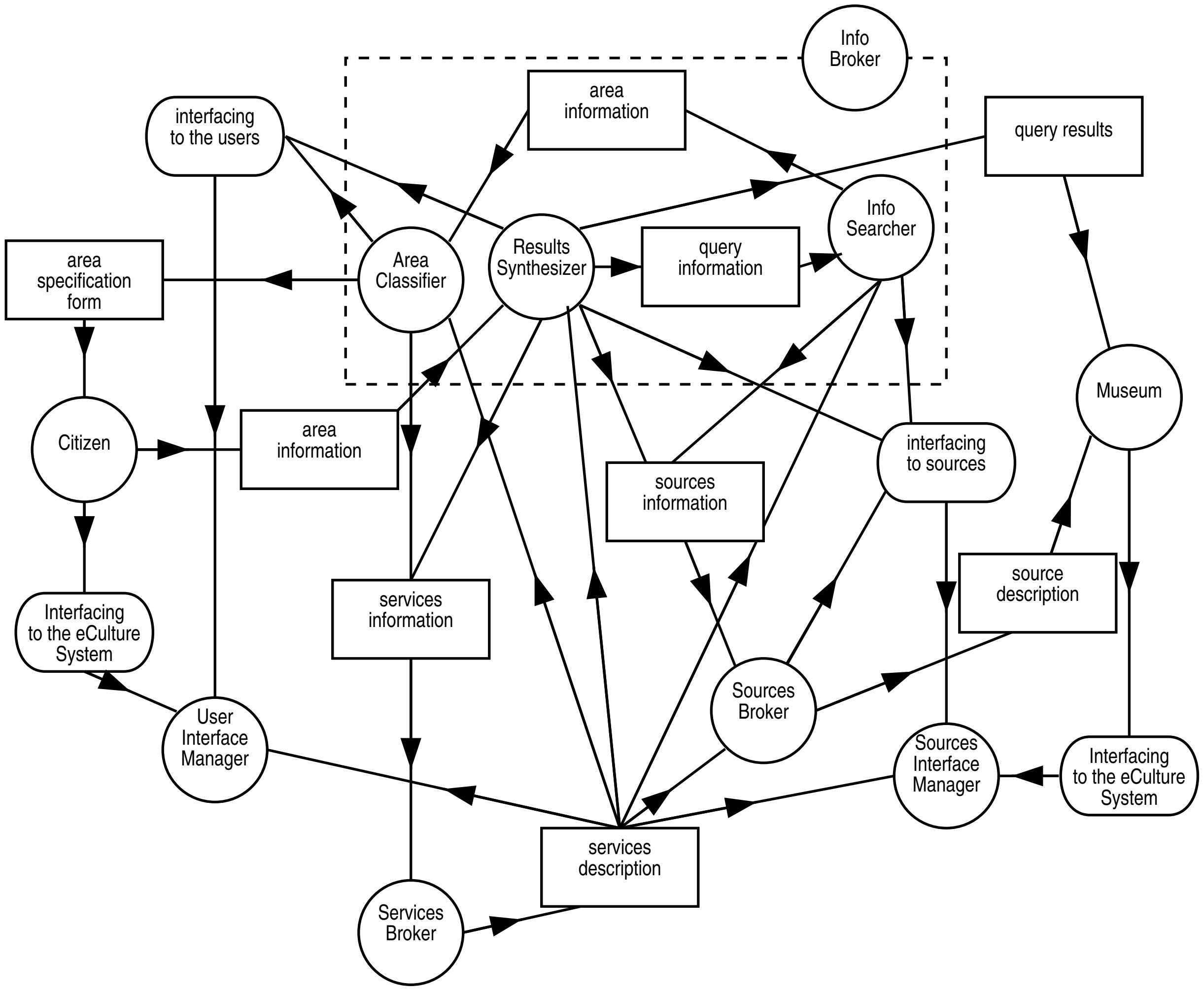


Figure 3.7: Extended actor diagram w.r.t. the Info Broker (step 1).

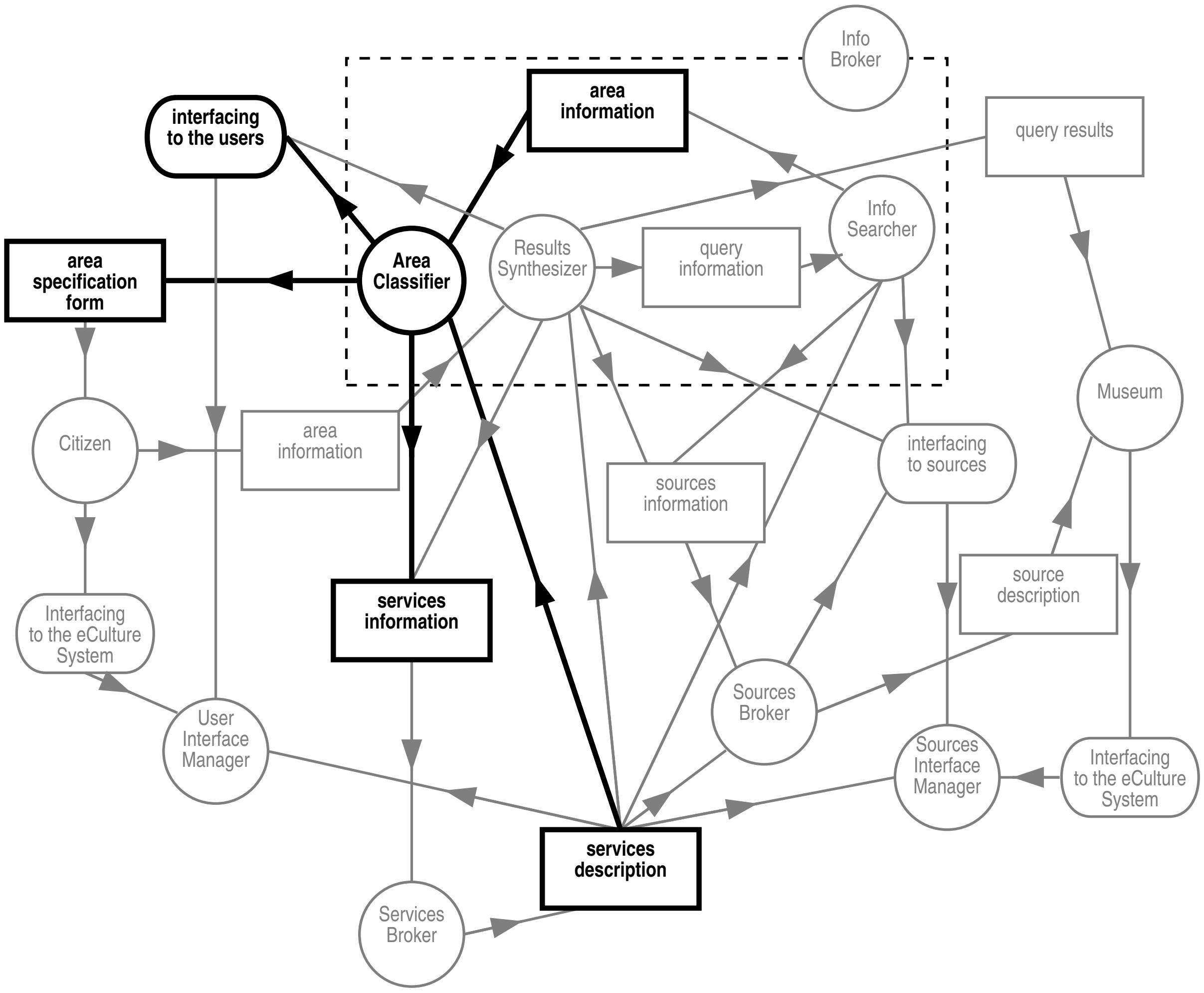


Figure 3.8: Identifying actor capabilities from actor dependencies w.r.t. the Area Classifier (step 2).

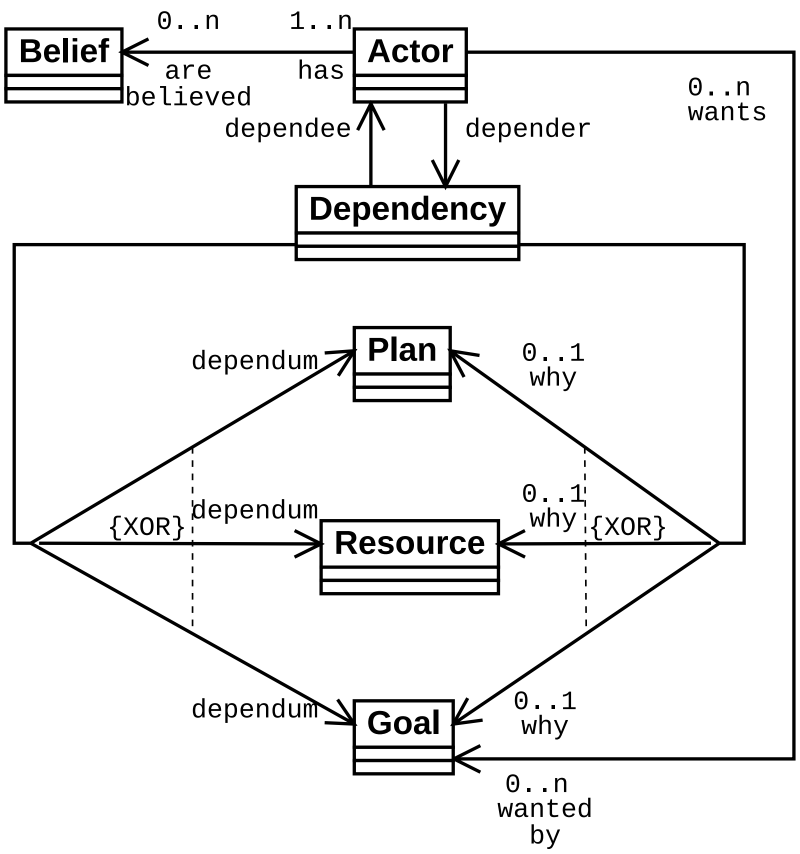


Figure 3.9: The UML class diagram specifying the actor concept in the Tropos meta-model.

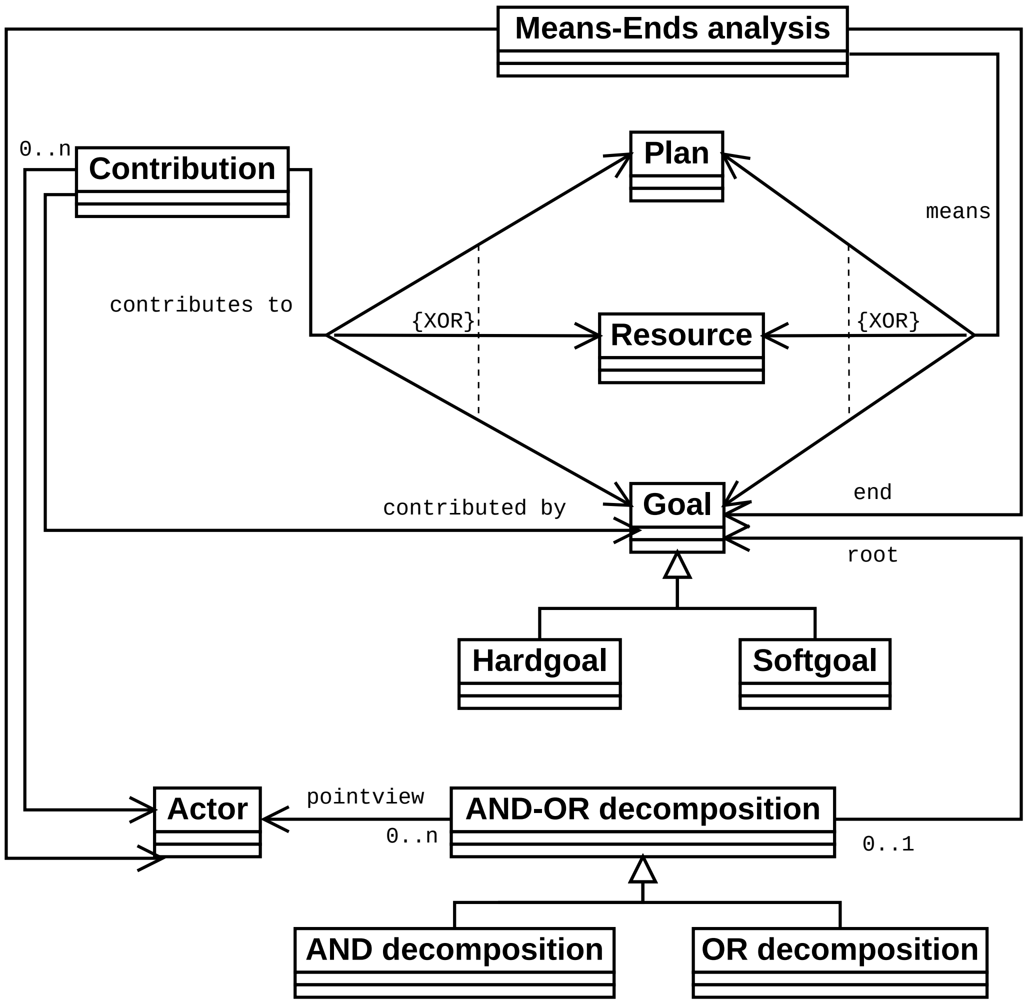


Figure 3.10: The UML class diagram specifying the the goal concept in the Tropos meta-model.

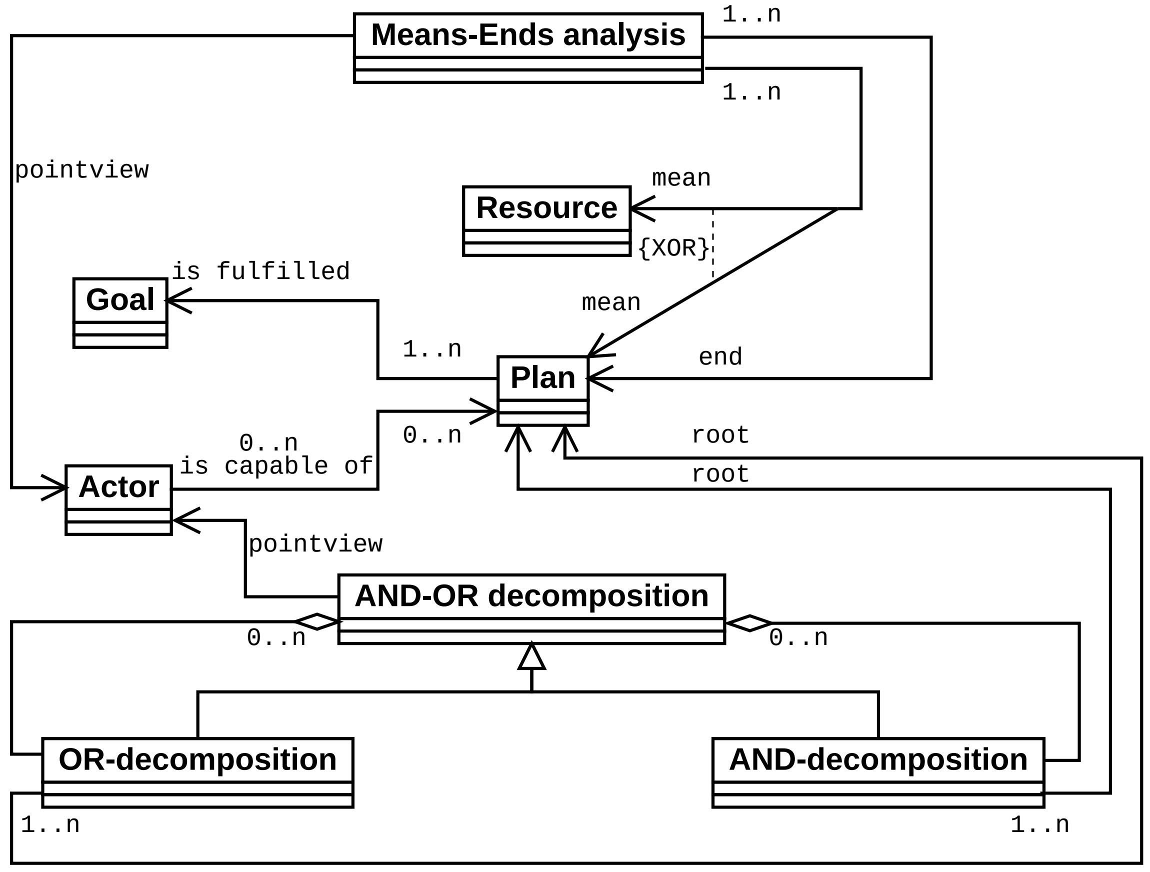


Figure 3.11: The UML class diagram specifying the plan concept in the Tropos meta-model.

Chapter 4

Integrated agent-oriented and UML methodologies for KLASE detailed design

4.1 The OPEN Process Framework and its Existing Agent oriented Enhancements

Integrating agent concepts into existing OO methodologies has resulted in several agent-oriented methodologies, for example, [DeL99b, CCE$^+$01, WJK00b, BGPG02b]. One which we will discuss is the OPEN Process Framework, or OPF [GHSY97, HSSY98, FHS02b], which is a little different from most others in that it offers a metamodel-underpinned framework rather than (strictly) a methodology.

Method engineering (e.g., [Bri96, THV97]) is then used to construct project-specific or "situational" methods (a.k.a. methodologies). This is possible because of the provision of a repository of method fragments (e.g., [vSH96]) or process components (e.g., [FHS02b]).

Initially, the repository of method fragments in OPEN was aimed at providing the ability to construct methodologies in the general area of information systems development. However, as new ideas emerged over the last few years, projects to extend the contents of the OPF repository have seen additions in areas such as component-based development [HS01], web-based development [HLHS02, HBL01] and organizational transition [HSS00]. Initial extensions to agent-oriented development were formulated in [DHS03b, HSD03] and it is these extensions which we evaluate for completeness against the agent-oriented Tropos methodology — a comparison which is the focus of this chapter.

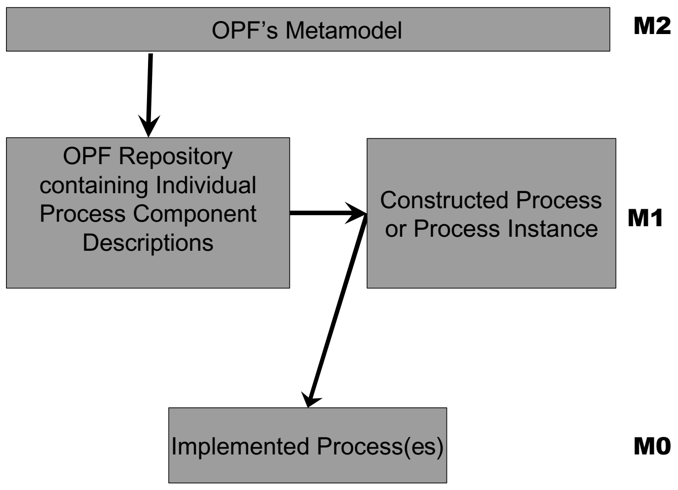

OPF's Metamodel
M2
OPF Repository containing Individual Process Component Descriptions
Constructed Process or Process Instance
M1
Implemented Process(es)
M0

Figure 4.1: Three metalevels (M2, M1, M0) that provide a framework in which the relationship between the metamodel (M2), the repository and process instances (M1) and the implemented process (M0) can be seen to co-exist.

4.2 The OPEN Process Framework

OPF consists of (i) a process metamodel or framework from which can be generated an organizationally-specific process (instance) created, using a method engineering approach [Bri96], from (ii) a repository and (iii) a set of construction guidelines. The metamodel can be said to be at the M2 metalevel (Figure 4.1) with both the repository contents and the constructed process instance at the M1 level (Figure 4.2). The M0 level in Figure 4.1 represents the execution of the organization's (M1) process on a single project. Each (M1) process instance is created by choosing specific process components from the OPF Repository (Figure 4.1) and constructing (using the Construction Guidelines) a specific configuration — the "personalized OO development process". Then, using this method engineering approach, from this process metamodel we can generate an organizationally-specific process (instance).

The major elements in the OPF metamodel are Work Units (Activities, Tasks and Techniques), Work Products and Producers [FHS02b] — see Figure 4.3. These three components interact; for example producers perform work units, work units maintain work products and producers produce work products. In addition to these three metatypes, there are two auxiliary ones (Stages and Languages), which interact as shown in Figure 4.3.

Activity is at the highest level in the sense that a process consists of a number of Activities. Activities are largescale definitions of *what* must be done. They are not used for

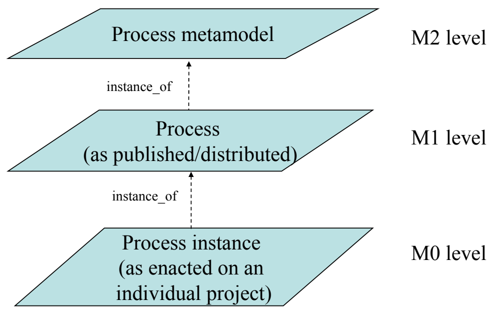


Figure 4.2: For process metamodelling, we adopt a 3 layer version of the 4 layer OMG UML metamodel [OMG01] in which every concept in layer x is an instance of a concept at level $x + 1$ (except for the self-referential top layer).

project management or enactment because they are at too high an abstraction level. Instead, OPEN offers the concept of Task (in agreement with the terminology of the Project Managers' Body of Knowledge [Dun96]) which is defined as being the smallest unit of work that can be project managed. Both Activities and Tasks are kinds of Work Unit in the OPF metamodel (Figure 4.3).

Work Products are the outputs of the Activities. These work products may be graphically or textually described. Thus, we need a variety of languages to describe them. Typical examples here are English (natural language), UML (modelling language) and C# (implementation language). Since the metamodel itself is a "design model", it is reasonable to document it with one of the available modelling languages. Here, we use the Unified Modeling Language of the OMG [OMG01] since it is probably the most commonly used (at least in OO developments).

While it is possible to analyze the metamodel directly, in this chapter we address the issue of whether the contents of the current repository for the OPF is adequate for supporting agent-oriented developments. This repository contains *instances* generated from each of the metaclasses in the metamodel. For each metaclass there are potentially numerous instances. These are documented in various books and papers, as noted earlier. The ones specific to agents are listed in Table 4.1 (see next section).

As a consequence of the modular nature of the OPEN approach to methodology, via the notion of a repository of process components together with the application of method or process engineering [RFKR00], it is relatively easy to add additional meta-elements and extremely easy to add additional examples of process components to the repository (as instances of pre-existing meta-elements). To extend this approach to support agent-oriented information systems, Debenham and Henderson-Sellers [DHS03b] analyzed the differences between agent-oriented and object-oriented approaches in order to be able to itemize and outline the necessary additions to the OPEN Process Framework's repository in the standard format provided in [HSSY98]. The focus of that work was primarily on instances of the meta-class WorkUnit useful for agent-oriented methodologies and

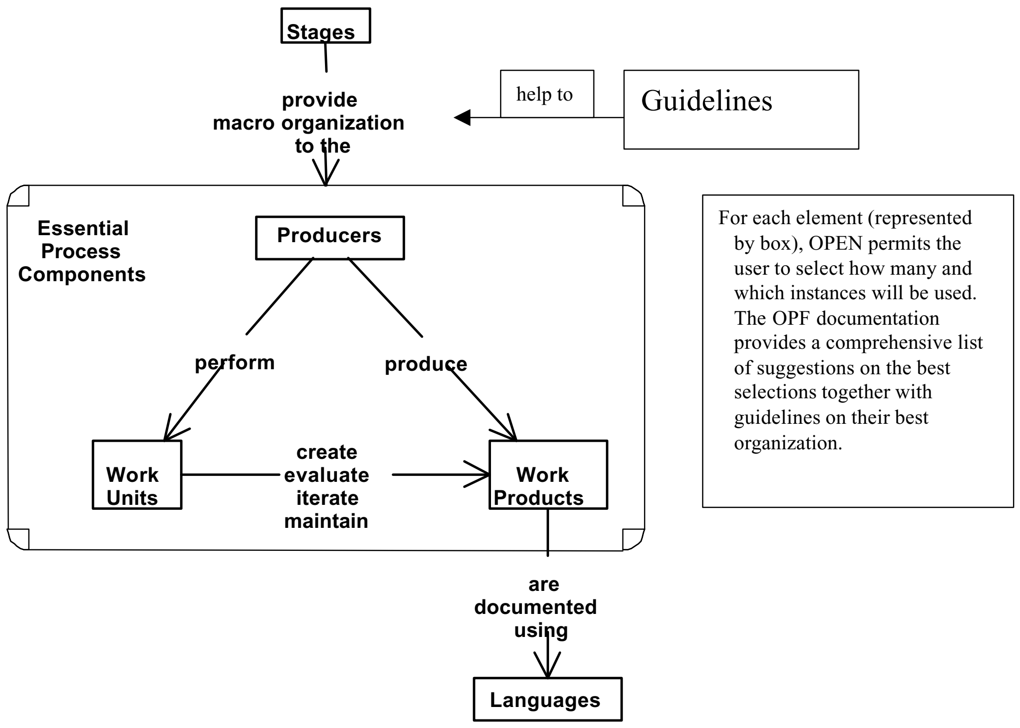

Stages
provide macro organization to the
help to
Guidelines
For each element (represented by box), OPEN permits the user to select how many and which instances will be used. The OPF documentation provides a comprehensive list of suggestions on the best selections together with guidelines on their best organization.
Essential Process Components
Producers
perform
produce
Work Units
create evaluate iterate maintain
Work Products
are documented using
Languages

Figure 4.3: The five major metatypes in the OPF metamodel (after [FHS02b]).

processes. Table 4.1 lists the Tasks and Techniques so far added to the OPF repository (no new Activities were identified).

4.3 Supporting Tropos Concepts in the OPEN Process Framework

In this section, we evaluate the existing Agent OPEN description (summarized in Section 2.2 above) against ideas formulated within the Tropos methodology, seeking any omissions or poor support of Tropos elements in the OPF. We then make recommendations for enhancements to the OPF in order that it can fully support all agent-oriented concepts formulated in Tropos.

Several new process components (method chunks) need to be added to the existing OPF repository. These are primarily Tasks and Techniques but there is also one new Activity: Early Requirements Engineering (in Tropos called the Early Requirements Analysis phase) as well as some work products. All of these are outlined below in standard OPEN format and summarized for convenience in Table 4.2.

Tasks for AOIS	**Techniques likely to be useful**
Identify agents' roles	Environmental evaluation
Model the agent's environment	Environmental evaluation
Identify system organization	Environmental evaluation
Determine agent interaction protocol Determine delegation strategy	Contract nets Market mechanisms
Determine agent communication protocol	FIPA KIF compliant language
Determine conceptual architecture Determine agent reasoning	3-layer BDI model Deliberative reasoning: Plans Reactive reasoning: ECA Rules
Determine control architecture	Belief revision of agents Commitment management Activity scheduling Task selection by agents Control architecture
Determine system operation Gather performance knowledge	Learning strategies for agents
Determine security policy for agents	[topic of future research]
Undertake agent personalization	Environmental evaluation User model incorporation
Identify emergent behaviour	[topic of future research]

Table 4.1: Tasks and Techniques already proposed [DHS03b, HSD03] for addition to the OPF repository in order to support the development of agent-oriented systems.

Activity	
Early requirements engineering	
Tasks	**Related Techniques**
Model actors	
Model capabilities for actors	Capabilities identification and analysis
Model dependencies for actors and goals	Contribution analysis
Model goals	Means–End Analysis Contribution Analysis AND/OR Decomposition
Model plans	Means–End Analysis Contribution Analysis AND/OR Decomposition
Work Products	
(Tropos) actor diagram (Tropos) capability diagram (Tropos) goal diagram (Tropos) plan diagram	

Table 4.2: Activity, Tasks, Techniques and Work Products proposed for inclusion in the OPF repository as a result of analyzing Tropos.

4.3.1 Activity

An Activity in the OPF describes a coarse granular "job to be done". It describes "what" needs to be done but not "how". One new Activity is proposed here for inclusion in the OPF repository based on contributions made by Tropos.

Early requirements engineering focusses on domain modeling. It consists of identifying and analyzing the relevant actors in organizations and their goals or intentions. These actors may correspond with the stakeholders but may also include other social elements (individuals, but also organizations, organizational units, teams, and so on) who do not directly share an interest in the project, but still need to be modelled in order to produce a sufficiently complete picture of the organizational domain. Each organization active element is modelled as a (social) actor that is dependent upon another (social) actor in order for them to achieve some stated goal. During Early Requirements Engineering, these goals are decomposed incrementally and finally the atomic goals can be used to support an objective analysis of alternatives.

The results of this analysis can be documented using a variety of Tropos diagrams. Goals, actors and dependencies can be depicted on an actor diagram and, in more detail, on a goal diagram. These results then form the basis for the "late requirements analysis" which in OPEN is called simply Requirements Engineering in which the system requirements are elicited in the context of the stakeholders' goals identified in this activity of Early Requirements Engineering.

4.3.2 Tasks

A Task in the OPF describes a granular “job to be done”. As with Activities, a Task describes what is to be don but not how. However, the granularity is at an individual developer’s scale, in comparison to the team scale of an Activity. In the remaining subsections, we describe five new/modified Tasks in the layout style used as standard in the OPEN literature (e.g. [GHSY97]).

Task: Model actors

Focus: People, other systems and roles involved *Typical supportive techniques:* Business process modelling, Soft systems analysis *Explanation.* While the concept of actors in OO systems already exists (and is supported in the original OPF), the Tropos methodology extends the OO notion of an actor beyond that of a single person/system/role interacting with a system to that of a more general entity that has strategic goals and intentionality within the system or organizational setting [BGG^{+}03] including also, for example, whole organizations, organizational units and teams. Actors in Tropos can represent either agents (both human and artificial) or roles or positions (a set of roles, typically played by a single agent). This new Task thus considerably extends the existing concepts related to traditional OO actors. To model an actor, one must identify and analyze actors of both the environment and the system (or system-to-be). Tropos encourages the use of this Task in the early requirements phase for the modelling of domain stakeholders and their intentions as social actors. Actors can be depicted using (Tropos) actor diagrams (see below).

Task: Model capabilities for actors

Focus: Capability of each actor in the system *Typical supportive techniques:* Capabilities identification and analysis *Explanation.* The capability of an actor represents its ability to define, choose and execute a plan (for the fulfilment of a goal), given specific external environmental conditions and a specific event [BGG^{+}03]. Capability modelling commences after the architecture has been designed, subsequent to an understanding of the system sub-actors and their interdependencies. Each system subactor must be provided with its own individual capabilities, perhaps with additional “social capabilities” for managing its dependencies with other actors/subactors. Previously modelled goals and plans generally now become an integral part of the capabilities. Capabilities can be depicted using (Tropos) capability diagrams and plan diagrams (see below).

Task: Model dependencies for actors and goals

Focus: How/if an actor depends on another for goal achievements *Typical supportive techniques:* Contribution analysis, Delegation analysis *Explanation.* In Tropos, a dependency may exist between two actors so that one actor depends in some way on the other in order to achieve its own goal, a goal that cannot otherwise be achieved or not as well or as easily without involving this second actor. Similarly, a dependency between two actors may exist for plan execution or resource availability [BGG^{+}03]. The actors are named, respectively, the depender and the dependee while the dependency itself centres around the dependum. Dependencies can be depicted using (Tropos) actor diagrams and, in more detail, in goal diagrams (see below).

Task: Model goals

Focus: Actor's strategic interests *Typical supportive techniques:* Means-end analysis, contribution analysis, AND/OR decomposition *Explanation.* A goal represents an actor's strategic interests [BGG^{+}03] — Tropos recommends both hard and soft goals. Modelling goals requires the analysis of those actor goals from the view point of the actor itself. The rationale for each goal relative to the stakeholder needs to be analyzed — typical Techniques are shown in Table 4.2. Goals may be decomposed into subgoals, either as alternatives or as concurrent goals. Plans may also be shown together with their decomposition, although details of plans are shown in a Plan Diagram (q.v.). Goals can be depicted using (Tropos) goal diagrams (see below).

Task: Model plans

Focus: Means to achieve goals *Typical supportive techniques:* Means-end analysis, contribution analysis, AND/OR decomposition *Explanation.* A plan represents a means by which a goal can be satisfied or, in the case of a soft goal, satisficed [BGG^{+}03]. Plan modelling complements goal modelling and rests on reasoning techniques analogous to those used in goal modelling. Plans can be depicted using (Tropos) goal diagrams and plan diagrams (see below).

4.3.3 Techniques

To complement the "what" of Activities and Tasks, OPF Techniques detail "how" they are to be achieved. We have identified four new Techniques from Tropos and describe them here in the standard format for OPF Techniques [HSSY98].

Technique: Means–End Analysis

Focus: Identifying means to achieve goals *Typical tasks for which this is needed:* Model goals, Model plans *Description.* Means-end analysis aims at identifying plans, resources and goals as well as means to achieve the goals. *Usage.* To perform means-end analysis, the following are performed iteratively:

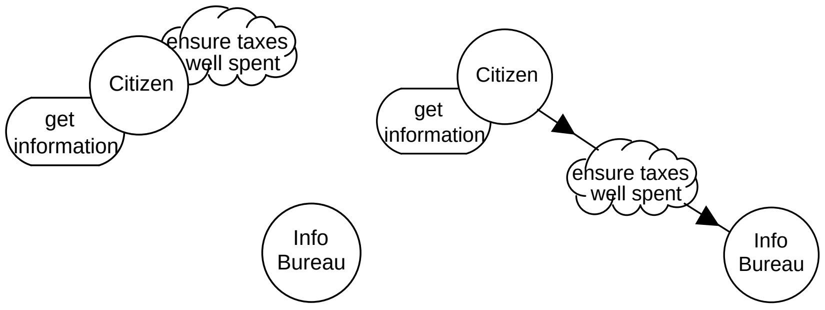
- Describe the current state, the desired state (the goal) and the difference between the two
- Select a promising procedure for enabling this change of state by using this identified difference between present and desired states.
- Apply the selected procedure and update the current state.

If this successfully finds an acceptable solution, then the iterations cease; otherwise they continue. If no acceptable solution is possible, then failure is announced.

Technique: Contribution Analysis

Focus: Goals contributing to other goals *Typical tasks for which this is needed:* Model goals, Model plans *Description.* Contribution analysis identifies goals that may contribute to the (partial) fulfilment of the final goal. It may be alternatively viewed as a kind of means-end analysis in which the goal is identified as the means [BGG+03]. Contribution analysis applied to soft-goals is often used to evaluate non-functional requirements. *Usage.* Identify goals and soft-goals that can contribute either positively or negatively towards the achievement of the overall goal or soft-goal. Of course the focus is on identifying positive contributions, but the technique may also lead, as a side effect, to the identification of negative contributions. Annotate these appropriately (say with + or −). A + label indicates a positive, partial contribution to the fulfilment of the goal being analyzed. Contribution analysis is very effective for soft goals used for eliciting non-functional (quality) requirements.

Technique: AND/OR Decomposition



(a) (b)

Figure 4.4: (a) Example actor diagram showing goals attached to actors; (b) Example actor diagram showing an explicit dependee, depender and dependum

Focus: Goal decomposition *Typical tasks for which this is needed:* Model goals, Model plans *Description*. This is a technique to decompose a root goal into a finer goal structure. *Usage*. Start with a high level goal and decompose into subgoals. These subgoals may either be alternatives (OR decomposition) or additive (AND decomposition).

Technique: Capabilities identification and analysis

Focus: Capabilities identification *Typical tasks for which this is needed:* Model capabilities for actors *Description*. For each goal introduced, we identify a set of capabilities that the responsible actor should have in order to fulfill the goal. When the achievement of the goal involves other actors, the analysis is expanded also to these actors. Capabilities for the interaction/collaboration are then identified and analyzed contextually(see [BGG$^+$03] for more details).*Usage*. Start with a goal associated to an actor and identify the capabilities needed locally. If the goal involves other actors the analysis is extended to these actors with respect to their contribution in the achievement of the goal.

4.3.4 Work Products

OPF Work Products describe artefacts that are created, consumed and/or maintained. Some of these act as deliverables, either to other team members or to a third party client. Four new work products are identified and described here.

Work Product: (Tropos) Actor Diagram

In Tropos, the actor diagram graphically depicts actors (as circles), their goals (as ellipses and clouds) attached to the relevant actor (Figure 4.4-a) together with a network of de-

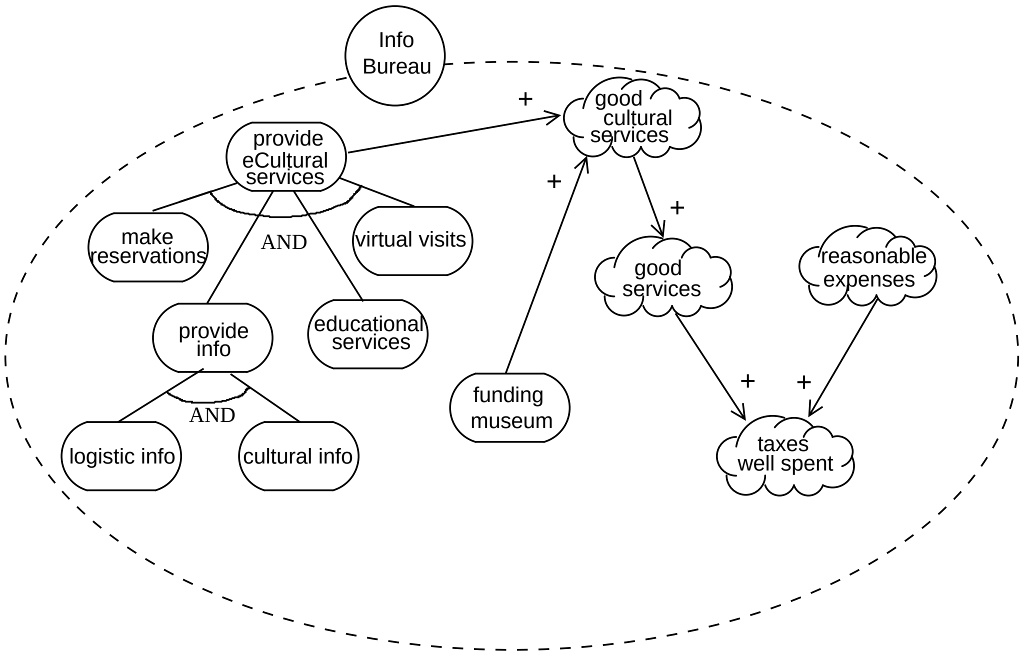


Figure 4.5: Example goal diagram (after [BGG$^+$03])

pendencies between the actors (Figure 4.4-b). In Figure 4.4-a, Citizen has two goals: the hard goal to "get information" and the soft goal to "ensure taxes well spent". However, this soft goal is best delegated to the Information Bureau actor. To show this delegation, the delegated goal is shown explicitly as a dependum (cloud symbol) connected by two line segments to the two actors (Citizen and Information Bureau) (Figure 4.4-b).

Work Product: (Tropos) Capability Diagram

A capability diagram is drawn from the viewpoint of a specific agent. They are initiated by an event caused by an external event. Nodes in the diagram model plans (which can be expanded through the use of a Plan Diagram (q.v)) and transition arcs model events. Beliefs are modelled as objects [BGG$^+$03]). Each node in the capability diagram may be expanded into a Plan Diagram (q.v.). Capability diagrams in Tropos use UML activity diagrams.

Work Product: (Tropos) Goal Diagram

Figure 4.5 shows an example goal diagram in which the focus is that of how Information Bureau tries to achieve the delegated soft-goal "taxes well spent". Providing good services with reasonable expenses, Information Bureau can contribute to spend taxes well. Good services may include good cultural services, which in turn may include services available via the web. So “provide eCultural services” can contribute positively in achieving the sotfgoal “good cultural services”. Figure 4.5 shows also the partial AND decomposition of the “provide eCultural services” goal.

Work Product: (Tropos) Plan Diagram

A plan diagram depicts the internal structure of a plan, summarized as a single node on a Capability diagram (q.v.). Plan diagrams in Tropos use UML activity diagrams.

4.4 Detailed design with AUML

The detailed design phase deals with the specification of the agents’ micro level. Agents’ goals, beliefs, and capabilities, as well as communication among agents are specified in detail. Practical approaches for this activity are usually proposed within specific development platforms and depend on the features of the adopted agent programming language. In other words, this step is usually strictly related to implementation choices. Moreover, the Object Management Group (OMG) and the Foundation for Intelligent Physical Agents (FIPA) are supporting the extension of the Unified Modeling Language (UML) [BRJ99] as the language which should enable the specification of agent systems [BMO01]. Agent UML packages modelling well-known agent communication protocols, such as the Contract Net, are already available [OPB00].

In Tropos, we adapt practical results from these approaches to agent systems design, but, despite them, we face the detailed design step on the basis of the specifications resulting from the architectural design phase and the reasons for a given element, designed at this level, can be traced back to early requirement analysis.

During detailed design, we use UML activity diagrams for representing capability and plan, and we adopt a subset of the AUML diagrams proposed in [OPB00] for specifying agents protocols.

***C*apability diagrams.** The UML activity diagram allows us to model a capability (or a set of correlated capabilities) from the point of view of a specific agent. External events set up the starting state of a capability diagram; activity nodes model plans, transition arcs model events, and beliefs are modeled as objects. For instance, Figure 4.6 depicts the capability diagram of the present query results capability of the User Interface Agent.

***P*lan diagrams.** Each plan node of a capability diagram can be further specified by UML activity diagrams. For instance, Figure 4.7 depicts the plan evaluate query results belonging to the the capability depicted in the diagram of Figure 4.6. The plan evaluate

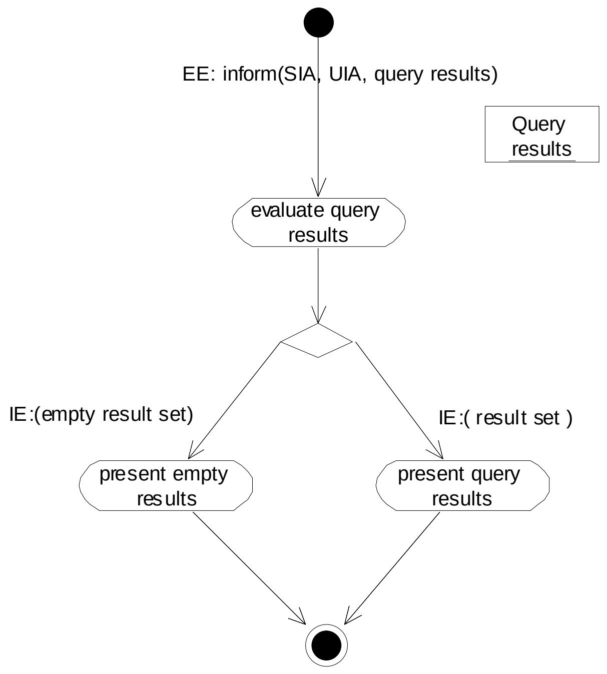


Figure 4.6: Capability diagram represented as an AUML activity diagram.

query results is activated by the arrival of the query results from the Synthesizer, and it ends storing an empty or no empty result set. Query results are compared to a set of possible result models contained into the agent's beliefs. Possible errors during the comparison yield the end of the plan without any storing. If there are no errors, the plan ends successfully storing a result set conform to the found result model. The plan can end successfully also when there are no result models comparable to the query results. In this case, the agent stores an empty result set.

Agent interaction diagrams. Here AUML sequence diagrams can be exploited. In AUML sequence diagrams, agents correspond to objects, whose life-line is independent from the specific interaction to be modeled (in UML an object can be created or destroyed during the interaction); communication acts between agents correspond to asynchronous message arcs.

Figure 4.8 shows a simple part of the communicative interaction among the system's agents and the user. In particular, the diagram models the interaction among the user (citizen), the User Interface Agent (UI), the Directory Facilitator (DF), and the Query Handler (QH). The interaction starts with an info request by the user to the UI, and ends with the results presentation by the UI to the user. The UI asks the user for the query

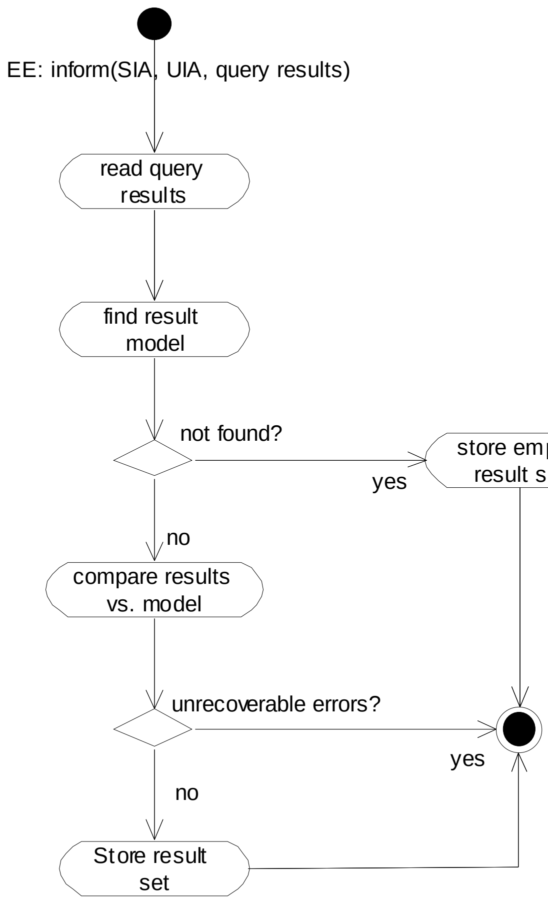


Figure 4.7: Plan diagram for the plan evaluate query. Ovals correspond to simple or complex actions, arcs to transitions from an action to the subsequent one, start and end states transitions to events.

specifications, and when the user replays, the UI asks the DF for the address of an agent able to provide the requested service. The DF sends the QH address to the UI so that the UI can ask the QH for the service. Finally, the QH sends the results to the UI, and then the UI presents the results to the user. The template packages of sequence diagrams, proposed in [OPB00] for modeling Agent Interaction Protocols, can be straightforwardly applied to our example. In such a case, each communicative act of Figure 4.8 must be analyzed in detail.

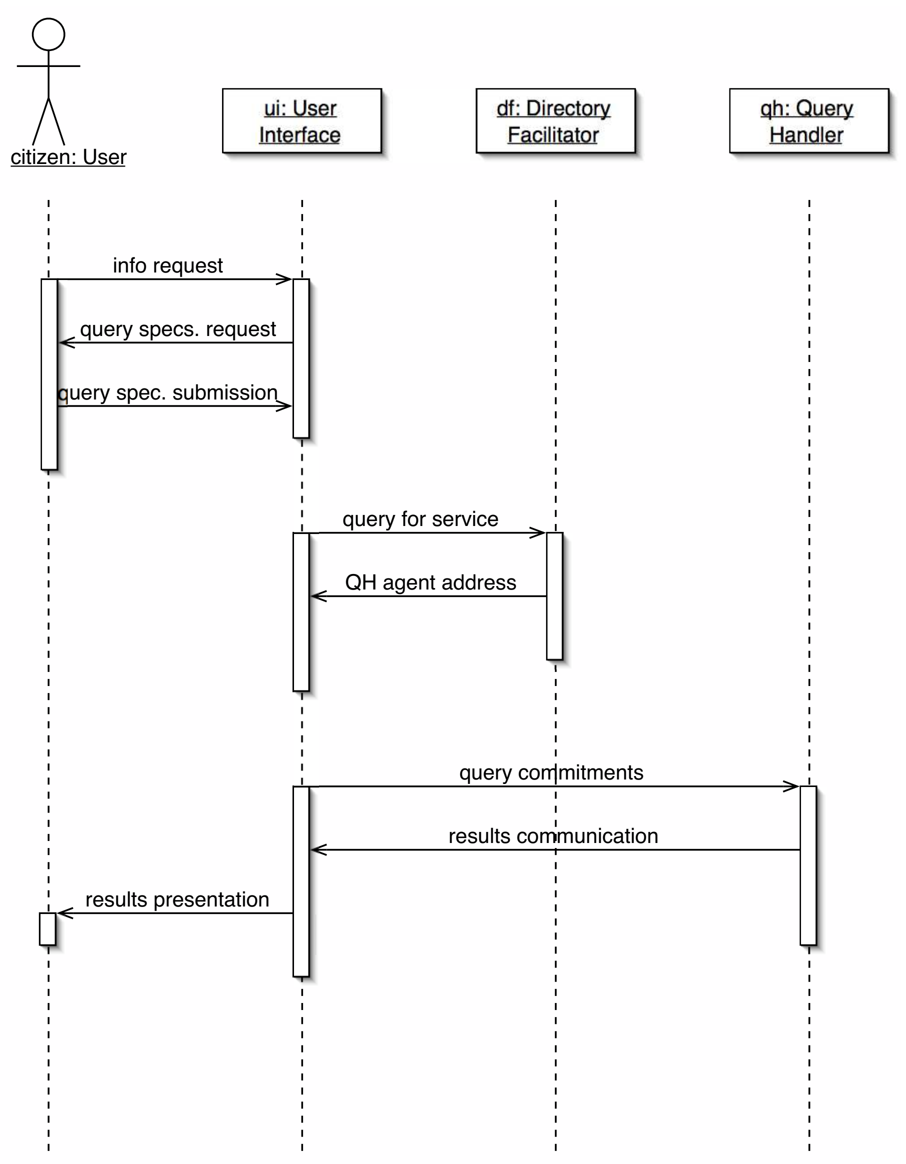


Figure 4.8: Agent interaction diagram. Boxes represent agents and arrows model communicative acts.

4.5 Implementation Using JACK

The BDI platform chosen for the implementation is JACK Intelligent Agents [Cob00a], an agent-oriented development environment built on top and fully integrated with Java. Agents in JACK are autonomous software components that have explicit goals (desires) to achieve or events to handle. Agents are programmed with a set of plans in order to make them capable of achieving goals. The implementation activity follows step by step, in a natural way, the detailed design specification described in Section 4.4. In fact, the notions introduced in that section have a direct correspondence with the following JACK's constructs, as explained below:

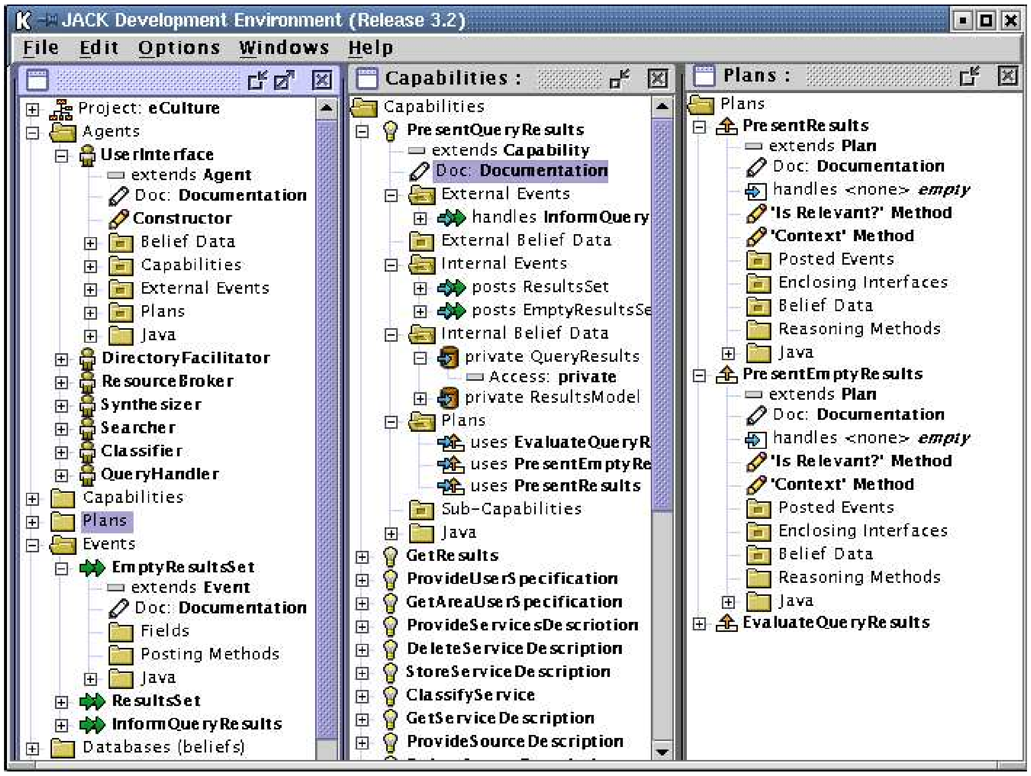
- *Agent*. A JACK's agent construct is used to define the behavior of an intelligent software agent. This includes the capabilities an agent has, the types of messages and events it responds to and the plans it uses to achieve its goals.
- *Capability*. A JACK's capability construct can include plans, events, beliefs and other capabilities. An agent can be assigned a number of capabilities. Furthermore, a given capability can be assigned to different agents. JACK's capability provides a way of doing reuse.
- *Belief*. The JACK's database construct provides a generic relational database. A database describes a set of beliefs that the agent can have.
- *Event*. Internal and external events specified in the detailed design map to the JACK's event construct. In JACK an event describes a triggering condition for agents actions.
- *Plan*. The plans contained into the capability specification resulting from the detailed design level map to the JACK's plan construct. In JACK a plan is a sequence of instructions the agent follows to try to achieve goals and handle designed events.

Figure 4.9 depicts the JACK layout presenting the eCulture System analyzed in the previous sections. The first window focuses on the declaration of the five agents, and in particular on the User Interface Agent and its capabilities. The definition for the User Interface Agent is as follows:

```
public agent UserInterface extends Agent {
    #has capability GetQueryResults;
    #has capability ProvideUserSpecification;
    #has capability GetUserSpecification;
    #has capability PresentQueryResults;
    #handles event InformQueryResults;
    #handles event ResultsSet; }
```

The second window lists all the capabilities associated to the agents of the system. The capability present query results, analyzed in Figure 4.6, is defined as follows:

```
public capability PresentQueryResults  extends Capability {
    #handles external event InformQueryResults;
    #posts event ResultsSet ;
    #posts event EmptyResultsSet ;
```


JACK Development Environment (Release 3.2)

File Edit Options Windows Help

```
Project: eCulture
Agents
  UserInterface
    extends Agent
    Doc: Documentation
    Constructor
    Belief Data
    Capabilities
    External Events
    Plans
    Java
  DirectoryFacilitator
  ResourceBroker
  Synthesizer
  Searcher
  Classifier
  QueryHandler
Capabilities
Plans
Events
  EmptyResultsSet
    extends Event
    Doc: Documentation
    Fields
    Posting Methods
    Java
  ResultsSet
  InformQueryResults
Databases (beliefs)
```

```
Capabilities :
Capabilities
  PresentQueryResults
    extends Capability
    Doc: Documentation
    External Events
      handles InformQuery
    External Belief Data
    Internal Events
      posts ResultsSet
      posts EmptyResultsSe
    Internal Belief Data
      private QueryResults
        Access: private
      private ResultsModel
    Plans
      uses EvaluateQueryR
      uses PresentEmptyRe
      uses PresentResults
    Sub-Capabilities
    Java
  GetResults
  ProvideUserSpecification
  GetAreaUserSpecification
  ProvideServicesDescriotion
  DeleteServiceDescription
  StoreServiceDescription
  ClassifyService
  GetServiceDescription
  ProvideSourceDescription
```

```
Plans :
Plans
  PresentResults
    extends Plan
    Doc: Documentation
    handles <none> empty
    'Is Relevant?' Method
    'Context' Method
    Posted Events
    Enclosing Interfaces
    Belief Data
    Reasoning Methods
    Java
  PresentEmptyResults
    extends Plan
    Doc: Documentation
    handles <none> empty
    'Is Relevant?' Method
    'Context' Method
    Posted Events
    Enclosing Interfaces
    Belief Data
    Reasoning Methods
    Java
  EvaluateQueryResults
```

Figure 4.9: JACK Developing Environment for the eCulture project.

```
    #private database QueryResults ();
    #private database ResultsModel ();
    #uses plan EvaluateQueryResults;
    #uses plan PresentEmptyResults;
    #uses plan PresentResults; }
```

The last window presents the plans associated to the capability present query results. The plan evaluate query results, analyzed in detail in the previous section (i.e., the plan evaluate query described in the plan diagram of Figure 4.7), is defined as follows:

```
public plan EvaluateQueryResults extends Plan {
    #handles event InformQueryResults ev;
    static boolean relevant(InformQueryResults ev) {return true}
    static model md;
    static queryResults qr;
    body ()
    { if (readQueryResults(qr))
        { if (findResultModel(qr,md))
            { if(compareResultModel(md)) {storeResults(qr,md)} }
          else { storeEmptyResults(); }
        }
      else { System.err(1); }}}
```

Chapter 5

Conclusion

This deliverable discusses Tropos, a new agent oriented software development methodology which spans the software development process from early requirements to implementation for agent oriented software. The deliverable presents and discusses the five phases supported by Tropos, the development process within each phase, the models created through this process, and the diagrams used to describe these models.

Throughout, we have emphasized the uniform use of a small set of knowledge level notions during all phases of software development. We have also provided an iterative, actor and goal based, refinement algorithm which characterizes the refinement process during each phase. This refinement process, of course, is instantiated differently during each phase.

Our long term objective is to provide a complete and detailed account of the Tropos methodology. Object-oriented and structured software development methodologies are examples of the breadth and depth of detail expected by practitioners who use a particular software development methodology. Of course, much remains to be done towards achieving this goal. We are currently working on several open points, such as the development of formal analysis techniques for Tropos [FPMT01]; the formalization of the transformation process in terms of primitive transformations and refinement strategies [BPG$^+$01c]; the definition of a catalogue of architectural styles for multi-agent systems which adopt concepts from organization theory and strategic alliances literature [KGM01]; and the development of tools which support the methodology during particular phases.

We consider a broad coverage of the software development process as essential for agent-oriented software engineering. It is only by going up to the early requirements phase that an agent-oriented methodology can provide a convincing argument against other, for instance object-oriented, methodologies. Specifically, agent-oriented methodologies are inherently *intentional*, founded on notions such as those of agent, goal, plan, etc. Object-oriented ones, on the other hand, are inherently *not* intentional, since they are founded on implementation-level ontological primitives. This fundamental difference shows most clearly when the software developer is focusing on the (organizational) environment where the system-to-be will eventually operate. Understanding such an environment calls (more precisely, cries out) for knowledge level modeling primitives. The agent-oriented programming paradigm is the only programming paradigm that can gracefully and seamlessly integrate the intentional models of early development phases with implementation and run-time phases. This is the argument that justifies agent-oriented software development, and at the same time promises for it a bright future.

In analyzing the extent to which other methodological frameworks, and in particular the OPEN Process Framework, supports these ideas, many deficiencies were identified. Here, we have itemized these gaps in OPEN's repository of process components and proposed additions to the repository specifically to address activities, tasks, techniques and work products found in Tropos but, until now, not available in the OPF repository.

We intend to progress this cross-fertilization between OPEN and Tropos, specifically taking advantage of the strengths of each: the early requirements engineering and agent focus of Tropos and the full lifecycle process of OPEN together with its metamodel-based underpinning that permits it to be used for situated method engineering [Bri96].

Chapter 6

History of the Deliverable

Below we outline how the work described in the deliverable has evolved along the years of the project.

Change History

Version	Date	Status	Author (Unit)	Description
0.1	2003/Nov	Draft	ITC, UNITN	First version of Tropos, the KLASE proposed methodology.
0.2	2004/Nov	Draft	UNITN	First version of integrated AO and UML methodologies for KLASE.
0.3	2005/Nov	Draft	UNITN	First version of state of the art of AOSE w.r.t. Tropos.
1.0	2006/Nov	Final	UNITN	Final revision of the deliverable.

Bibliography

[BC02] P. Burrafato and M. Cossentino. Designing a multi-agent solution for a bookstore with the PASSI methodology. In *Procs. Agent-Oriented Information Systems (eds. P. Giorgini, Y. Lesprance, G. Wagner and E. Yu)*, pages 102–118, 2002.

[BGG$^+$03] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopolous, and A. Perini. Tropos: an agent-oriented software development methodology. *Journal of Autonomous Multi-Agent Systems*, 2003.

[BGG$^+$04a] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopolous, and A. Perini. Tropos: an agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BGG$^+$04b] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

[BGGM01] C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors. *Cooperative Information Systems, 9th International Conference - CoopIS 2001*, volume 2172 of *Lecture Notes in Computer Science (LNCS)*, Trento, Italy, September 5-7 2001. Springer-Verlag.

[BGHSW05] P. Bresciani, P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors. *Agent-Oriented Information Systems II*, volume 3508 of *Lecture Notes in Artificial Intelligence, LNAI*. Springer-Verlag, 2005.

[BGPG02a] C. Bernon, M.-P. Gleizes, G. Picard, and P Glize. The ADELFE methodology for an intranet system design. In *Agent-Oriented Information Systems 2002. Procs. AOIS-2002(eds. P. Giorgini, Y. Lesprance, G. Wagner and E. Yu)*, pages 1–15, 2002.

[BGPG02b] C. Bernon, P.-P. Gleizes, G. Picard, and P. Glize. The ADELFE methodology for an intranet system design. In *Procs. Agent-Oriented Information Systems 2002 (eds. P. Giorgini, Y. Lespérance, G. Wagner and E. Yu)*, Toronto, Canada, May 2002.

[BMO01] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[Boo94] G. Booch. *Object-Oriented Analysis and Design* . (second edition), The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, USA, 1994.

[BPG$^+$01a] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *5th international conference on autonomous agents (Agents 2001)*, pages 648–655, Montreal, 28 May -1 June 2001. ACM.

[BPG$^+$01b] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: a transformation based approach. In *Submitted to the Second International Workshop on AGENT-ORIENTED SOFTWARE ENGINEERING (AOSE-2001)*, Montreal, Canada, May 29th 2001.

[BPG$^+$01c] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: a transformation based approach. In Wooldridge et al. [WCW01].

[Bri96] S. Brinkkemper. Method engineering: engineering of information systems-bdevelopment methods and tools. *Inf. Software Technol*, 38(4):275–280, 1996.

[BRJ99] G. Booch, J. Rambaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[CAB$^+$94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, and H. Gilchrist. *Object-Oriented Development. The Fusion Method.* Prentice Hall, Englewood Cliffs, NJ, USA, 313pp, 1994.

[CCE$^+$01] G. Caire, P. Chainho, R. Evans, F. Garijo, Gomez Sanz, Kearney J., Leal P., Massonet F., Pavon P., J., and J. Stark. Agent-oriented analysis using MESSAGE/UML. In *Procs. Second Int. Workshop on Agent-Oriented Software Engineering (AOSE–2001)*, pages 101–107, Montreal, Canada, May 2001.

[CCG$^+$01] G. Caire, W. Coulier, F. Garijo, J. Gomez, Pavon, Leal J., Chainho F., Kearney P., Stark P., Evans J., R., and P Massonet. Agent oriented analysis using MESSAGE/UML. In *Agent-Oriented Software Engineering II (eds. M. Wooldridge, G. Wei and P. Ciancarini),LNCS 2222, Springer-Verlag, Berlin, Germany, 119-135*, pages 119–135, 2001.

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), March 2000.

[CD98] A. Collinot and A. Drogoul. Using the Cassiopeia method to design a soccer robot team. *Applied Articial Intelligence (AAI) Journal*, 12(2-3):127–147, 1998.

[CDB96] A. Collinot, A. Drogoul, and P. Benhamou. Agent oriented design of a soccer robot team. In *Procs. Second Intl. Conf. on Multi-Agent Systems (ICMAS'96)*, 1996.

[CDJT00] C. Castelfranchi, F. Dignum, C. Jonker, and J. Treur. Deliberate normative agents: Principles and architectures. In *Intelligent Agents VI (eds N. Jennings and Y. Lesprance)*, pages 364–378. Springer-Verlag, Berlin, Germany, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[CL91] P.R. Cohen and H.J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.

[CNYM00] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[Cob00a] M. Coburn. JACK Intelligent Agents User Guide. AOS Technical Report, Agent Oriented Software Pty Ltd, July 2000. http://www.jackagents.com/docs/jack/html/index.html.

[Cob00b] M. Coburn. JACK Intelligent Agents User Guide . Technical report, Agent Oriented Software Pty Ltd, http://www.jackagents.com/docs/jack/html/index.html, July 2000.

[CP02] M. Cossentino and C. Potts. A CASE tool supported methodology for the design of multi-agent systems. *The 2002 International Conference on Software Engineering Research and Practice (SERP'02)*, 2002.

[CR02] L. Cernuzzi and G. Rossi. On the evaluation of agent oriented methodologies. In *Procs. OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 21–30. Centre for Object Technology Applications and Research, Sydney, 2002.

[CS98] L. Cavedon and L. Sonenberg. On social commitment, roles and preferred goals. In *International Conference on Multi-Agent Systems (ICMAS)*, 1998.

[DeL99a] S.A. DeLoach. MultiAgent Systems Engineering: a methodology and language for designing agent systems. *Procs AOIS*, 1999.

[DeL99b] S.A. DeLoach. Multiagent systems engineering: a methodology and language for designing agent systems. In *Procs. Agent-Oriented Information Systems '99 (AOIS'99)*, Seattle, WA, USA, 1 May 1999.

[DFM$^+$06] S. Dehousse, S. Faulkner, H. Mouratidis, M. Kolp, and P. Giorgini. Reasoning about willingness in networks of agents. In *Proceedings of the Fifth international workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS06) - in conjunction with ICSE06*, Shanghai, China, 22-23 May 2006.

[DHS03a] J. Debenham and B. Henderson-Sellers. Designing agent-based process systems - extending the OPEN Process Framework. In *Intelligent Agent Software Engineering (ed. V. Plekhanova)*, pages 160–190. Idea Group Publishing, Hershey, PA, USA, 2003.

[DHS03b] J. Debenham and B. Henderson-Sellers. *Intelligent Agent Software Engineering (ed. V. Plekhanova)*, chapter Designing agent-based process systems — extending the OPEN Process Framework, pages 160–190. Idea Group Publishing, 2003.

[DLF93] A. Dardenne, A. v. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[Dun96] W.R. Duncan. *A Guide to the Project Management Body of Knowledge*. Project Management Institute, PA, USA, 1996.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.

[DW04] K.H. Dam and M. Winikoff. Comparing agent-oriented methodologies. In *Agent-Oriented Systems (eds. P. Giorgini, B. Henderson-Sellers and M. Winikoff)*, pages 78–93. LNAI 3030, Springer-Verlag, Berlin, 2004.

[FGKM01] A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. Information Systems as Social Structures. In *Second International Conference on Formal Ontologies for Information Systems (FOIS-2001)*, Ogunquit, USA, October 17-19 2001.

[FHS02a] D.G. Firesmith and B. Henderson-Sellers. *The OPEN Process Framework*. Addison Wesley, Harlow, UK, 2002.

[FHS02b] D.G. Firesmith and B. Henderson-Sellers. *The OPEN Process Framework. An Introduction*. Addison-Wesley, Harlow, UK, 2002.

[FPMT01] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specification in Tropos. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering*, Toronto, CA, August 2001.

[GCL$^+$06] A. Garcia, R. Choren, C. Lucena, A. Romanovsky, T. Holvoet, and P. Giorgini, editors. *Software Engineering for Multi-Agent Systems IV*. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2006. In press.

[GGMS02] M. Garzetti, P. Giorgini, J. Mylopoulos, and F. Sannicoló. Applying tropos methodology to a real case study: Complexity and criticality analysis. In *workshop on Ďagli OGGETTI agli AGENTI - Dallínformazione alla Conoscenza (WOA02)*, Milano, Italy, November 2002.

[GHS05] P. Giorgini and B. Henderson-Sellers. Agent-oriented methodologies: an introduction. In *Agent-Oriented Methodologies*. Idea group, 2005.

[GHSW04] P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors. *Agent-Oriented Information Systems*, volume 303 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer-Verlag, 2004.

[GHSY97] I. Graham, B. Henderson-Sellers, and H. Younessi. *The OPEN Process Specification*. Addison-Wesley, Harlow, UK, 1997.

[GKMC05] P. Giorgini, M. Kolp, J. Mylopoulos, and J. Castro. Tropos: A requirements-driven methodology for agent-oriented software. In *Agent-Oriented Methodologies*. Idea group, 2005.

[GKMP04a] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. The Tropos methodology: an overview. In *Methodologies And Software Engineering For Agent Systems (eds. F. Bergenti, M.P. Gleizes and F. Zambolli)*. Kluwer Academic Publishing, 2004.

[GKMP04b] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. The tropos methodology: An overview. In *Methodologies and Software Engineering for Agent Systems*. Kluwer Academic Publishers, 2004.

[GLWY02a] P. Giorgini, Y. Lesperance, G. Wagner, and E. Yu, editors. *Agent-Oriented Information Systems I, Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information systems (AOIS-02) at CAiSE2002*, Toronto, Canada, 2002.

[GLWY02b] P. Giorgini, Y. Lesperance, G. Wagner, and E. Yu, editors. *Agent-Oriented Information Systems II, Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information systems (AOIS-02) at AAMAS2002*, Bologna, Italy, 2002.

[GMO04] P. Giorgini, J. P. Muller, and J. Odell, editors. *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2004.

[GMO05] P. Giorgini, J. P. Muller, and J. Odell, editors. *Agent-Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2005.

[GPM$^+$01a] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE01)*, Buenos Aires - ARGENTINA, June 13-15 2001.

[GPM$^+$01b] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. In S. Sen J.P. Müller, E. Andre and C. Frassen, editors, *Proceedings of the Thirteenth International Conference on Software Engineering - Knowledge Engineering (SEKE01)*, Buenos Aires - ARGENTINA, June 13 - 15 2001.

[HBL01] B. Haire, Henderson-Sellers B., and D. Lowe. Supporting web development in the OPEN process: additional tasks. In *Procs. 25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 383–389. IEEE Computer Society Press, Los Alamitos,CA, USA, 2001.

[HLHS02] B. Haire, D. Lowe, and B. Henderson-Sellers. Supporting web development in the OPEN process. In *Object-Oriented Information Systems (eds. Z. Bellahsène, D. Patel and C. Rolland)*. LNCS 2425, Springer–Verlag, 2002.

[HS95] B Henderson-Sellers. Who needs an OO methodology anyway? *J. Obj.-Oriented Programming*, 8(6):6–8, 1995.

[HS01] B. Henderson-Sellers. *An OPEN process for component-based development*, chapter Chapter 18. Component-Based Software Engineering: Putting the Pieces Together (eds. G.T. Heineman and W. Councill), Addison-Wesley, Reading, MA, USA, 2001.

[HSD03] B. Henderson-Sellers and J. Debenham. Towards OPEN methodological support for agent oriented systems development, 2003. submitted for publication.

[HSG05] B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea group, 2005.

[HSGB03a] B. Henderson-Sellers, P. Giorgini, and P. Bresciani. Enhancing agent OPEN with concepts used in the tropos methodology. In *Proceedings of the Fourth International Workshop Engineering Societies in the Agents World*, Imperial College London, UK, 29-31 October 2003.

[HSGB03b] B. Henderson-Sellers, P. Giorgini, and P. Bresciani. Evaluating the potential for integrating the open and tropos metamodels. In *Proccedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03)*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 23 - 26 2003.

[HSS00] B. Henderson-Sellers and M. Serour. Creating a process for transitioning to object technology. In *Proceedings Seventh Asia–Pacific Software Engineering Conference. APSEC 2000*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2000.

[HSSY98] B. Henderson-Sellers, A.J.H. Simons, and H. Younessi. *The OPEN Toolbox of Techniques*. Addison-Wesley, UK, 1998.

[HSTD05] B. Henderson-Sellers, Q.-N.N. Tran, and J. Debenham. An etymological and metamodel-based evaluation of the terms "goals and tasks" in agent-oriented methodologies. *J. Object Technol.*, 4, 2005.

[HSU00] B. Henderson-Sellers and B. Unhelkar. *OPEN Modeling with UML*. Addison-Wesley, London, 2000.

[Hug02] M.-Ph. Huget. Nemo: an agent-oriented software engineering methodology. In *Procs. OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 43–53. Centre for Object Technology Applications and Research, Sydney, Australia, 2002.

[IGGV96] C.A. Iglesias, M. Garijo, J.C. Gonzalez, and J.R. Velasco. A methodological proposal for multiagent systems development extending CommonKADS. In *Proc. of 10th KAW*, Banff, Canada, 1996.

[IGGV98] C.A. Iglesias, M. Garijo, J.C. Gonzalez, and J.R. Velasco. Analysis and design of multi-agent systems using MAS-CommonKADS. *Intelligent Agents IV: Agent Theories, Architectures, and Languages (eds. M.P. Singh, A. Rao and M.J. Wooldridge), LNAI Volume 1365, Springer-Verlag, Berlin, Germany*, 1998.

[IHK99] J. Iivari, R. Hirschheim, and H.K. Klein. Beyond methodologies: keeping up with information systems development approaches through dynamic classification. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS 1999)*, 1999.

[JAD] JADE: Java Agent Development Framework. 2004 on line documentation of JADE official site: http://jade.cselt.it/.

[Jay94] N. Jayaratna. *Understanding and Evaluating Methodologies, NISAD: A Systematic Framework*. McGraw-Hill, Maidenhead, Berks, UK, 1994.

[JMJ02] Castro J., Kolp M., and Mylopoulos J. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 27(6):365–389, 2002.

[Ken00] E.A. Kendall. Software engineering with role modelling. In *Procs. Agent-Oriented Software Engineering Workshop*, volume 1957, pages 163–169. LNCS,Springer-Verlag, Berlin, 2000.

[KGM01] M. Kolp, P. Giorgini, and J. Mylopoulos. An goal-based organizational perspective on multi-agents architectures. In *Proc. of the 8th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Seattle, WA, August 2001.

[KGR96] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling techniques for systems of BDI agents. *Technical Note 58, Australian Artificial Intelligence Institute, also published in Agents Breaking Away: Procs. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 56–71, 1996.

[KMJ96] E.A. Kendall, M.T. Malkoun, and C. Jiang. A methodology for developing agent based systems for enterprise integration. In *Modelling and Methodologies for Enterprise Integration (eds. P. Bernus and L. Nemes)*. Chapman and Hall, 1996.

[Kru99] Ph. Kruchten. *The Rational Unified Process. An Introduction*. Addison-Wesley, Reading, MA, USA, 1999.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. Addison-Wesley, Reading, MA, 2003.

[KZ98] E.A. Kendall and L. Zhao. Capturing and Structuring Goals. *Workshop on Use Case Patterns, Object Oriented Programming Systems Languages and Architectures*, 1998.

[Lin99] J. Lind. *Iterative Software Engineering for Multiagent Systems. The MASSIVE Method*. LNAI 1994, Springer-Verlag, Berlin, 1999.

[MCD$^+$01] E. Milgrom, P. Chainho, Y. Deville, Kearney Evans, R., P., and Ph. Massonet. MESSAGE: Methodology for Engineering Systems of Software Agents. Final Guidelines for the Identification of Relevant Problem Areas where Agent Technology Is Appropriate. *EUROSCOM Project Report P907*, 2001.

[MCN92] J. Mylopoulos, L. K. Chung, and B. A. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, June 1992.

[MGK02] J. Mylopoulos, P. Giorgini, and Kolp. Agent oriented software development. In *Proceedings of the 2nd Hellenic Conference on Artificial Intelligence (SETN-02)*, Greece, April 2002.

[MKFG05] H. Mouratidis, M. Kolp, S. Faulkner, and P. Giorgini. A secure architectural description language for agent systems. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS05)*, Utrecht-the Netherlands, July 2005. ACM Press.

[New82] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.

[Nwa96] H. Nwana. Software agents: An overview. *Knowledge Engineering Review Journal*, 11(3), November 1996.

[OMG99] OMG. *OMG Unified Modeling Language Specification*, version 1.3, alpha edition, January 1999.

[OMG01] OMG: OMG Unified Modeling Language Specification, Version 1.4. OMG document formal/01-09-68 through 80 (13 documents), September 2001. Available http://www.omg.org.

[OPB00] J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proc. of the Agent-Oriented Information Systems workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, Austin, TX, 2000.

[OVDPB00] J. Odell, H. Van Dyke Parunak, and B. Bauer. Extending UML for agents. In *Procs. Agent-Oriented Information Systems Workshop(eds. G. Wagner, Y. Lesperance and E. Yu)*, pages 3–17. 17th National Conference on Artificial Intelligence,, Austin, TX, USA, 2000.

[PBG$^+$01] A. Perini, P. Bresciani, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Towards an agent oriented approach to software engineering. *AI*IA Notizie : periodico dell'associazione italiana per l'intelligenza artificiale*, 2001.

[PGSF05] J. Pavon, J. Gomez-Sanz, and R. Fuentes. The INGENIAS methodology and tools. In *Agent-Oriented Methodologies (eds. B. Henderson-Sellers and P. Giorgini)*. Chapter 4. Idea Group, Hershey, PA, USA, 2005.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[RFKR00] C. Rupprecht, M. Fünffinger, H. Knublauch, and T. Rose. Capture and dissemiantion of experience about the construction of engineering processes. In *Procs. CAiSE 2000*. LNCS 1789, Springer Verlag, Berlin, 2000.

[RG91] A.S. Rao and M.P. Georgeff. Modelling rational agents within a BDI-architecture. In *Proceedings of Knowledge Representation and Reasoning (KRR-91) Conference*, San Mateo CA, 1991.

[RG95] A.S. Rao and M.P. Georgeff. BDI agents: from theory to practice. In *Procs. First International Conference on Multi Agent Systems*, pages 312–319, San Francisco, CA, USA, 1995.

[RP96] C. Rolland and N. Prakash. A proposal for context-specific method engineering. In *Procs. IFIP WG8.1 Conf. on Method Engineering*, pages 191–208. Chapman and Hall, 1996.

[RPB99] C. Rolland, N. Prakash, and A. Benjamen. A multi-model view of process modelling. *Requirements Eng. J.*, 4(4):169–187, 1999.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.

[SPG01] F. Sannicolò, A. Perini, and F. Giunchiglia. The Tropos modeling language. a User Guide. Technical report, ITC-irst, December 2001.

[SPMG05] A. Susi, A. Perini, J. Mylopoulos, and P. Giorgini. The tropos metamodel and its use. *Informatica, Slovene Society Informatika*, 29/4:377–443, November 2005.

[SS04] A. Sturm and O. Shehory. A framework for evaluating agent-oriented methodologies. In *Agent-Oriented Systems (eds. P. Giorgini, B. Henderson-Sellers and M. Winikoff)*, pages 94–109. LNAI 3030, Springer-Verlag, Berlin, 2004.

[SZ04] L. Shan and H. Zhu. CAMLE: a caste-centric agent-oriented modeling language and environment. In *Procs. Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, pages 24–25, Edinburgh, May 2004. Springer-Verlag.

[THV97] A.H.M. Ter Hofstede and T.F. Verhoef. On the feasibility of situational method engineering. *Information Systems*, 22:401–422, 1997.

[TLW04] Q.-N.N. Tran, G. Low, and M.-A. Williams. A preliminary comparative feature analysis of multi-agent systems development methodologies. In *Procs. AOIS@CAiSE*04*, pages 386–398. Faculty of Computer Science and Information, Riga Technical University, Latvia, 2004.

[vSH96] K. van Slooten and B. Hodes. Characterizing IS development projects. In *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support (eds. S. Brinkkemper, K. Lyytinen, R. Welke)*, pages 29–44. Chapman&Hall, Great Britain, 1996.

[Wag03] G. Wagner. The Agent-Object Relationship metamodel: towards a unified view of state and behaviour. *Inf. Systems*, 28(5):475–504, 2003.

[WCW01] M. Wooldridge, P. Ciancarini, and G. Weiss, editors. *Proc. of the 2nd Int. Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, CA, May 2001.

[WD00] M. Wood and S.A. DeLoach. An overview of the MultiAgent Systems Engineering methodology. In *Procs. 1st International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, pages 207–222, 2000.

[Wei99] G. Weiss, editor. *Multiagent System: a modern approach to Distributed AI*. MIT Press, 1999.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[WJK00a] M. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *J. Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.

[WJK00b] M. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *J. Autonomous Agents and Multi-Agent Systems*, 3:285–313, 2000.

[WPH01] M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In *Procs. 14th Australian Joint Conference on Artificial Intelligence (AI'01)*, Adelaide, 10-14 December 2001.

[WT05] G. Wagner and K. Taveter. Towards radical agent-oriented software engineering processes based on AOR modelling. In *Agent-Oriented Methodologies (eds. B. Henderson-Sellers and P. Giorgini)*. Chapter 10. Idea Group, Hershey, PA, USA, 2005.

[YM94] E. Yu and J. Mylopoulos. Understanding 'why' in software process modeling, analysis and design. In *Proceedings Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[YM96] E. Yu and J. Mylopoulos. Using goals, rules, and methods to support reasoning in business process reengineering. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 1(5), January 1996.

[YS02] P. Yolum and M.P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, 2002.

[Yu93] E. Yu. Modeling organizations for information systems requirements engineering. In *Proceedings of the First IEEE International Symposium on Requirements Engineering*, pages 34–41, San Jose, January 1993. IEEE.

[Yu95a] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.

[Yu95b] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.

[Yu01] E. Yu. Agent-oriented modeling: Software versus the world. In Wooldridge et al. [WCW01].

[ZJW01] F. Zambonelli, N. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In *Procs. Agent-Oriented Software Engineering Workshop*, volume 1957, pages 235–251. LNCS,Springer-Verlag, Berlin, 2001.

[ZJW03] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transaction on Software Engineering and Methodology*, 12(3):317–370, 2003.