UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

TROPOS: AN AGENT-ORIENTED SOFTWARE DEVELOPMENT METHODOLOGY

Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia,
John Mylopoulos, and Anna Perini

2002

Technical Report # DIT-02-0015

Also: submitted to the AAMAS Journal

TROPOS: An Agent-Oriented Software Development Methodology

Paolo Bresciani
ITC-Irst - Povo (Trento) - Italy - bresciani@irst.itc.it

Paolo Giorgini
Department of Information and Communication Technology
University of Trento - Italy - paolo.giorgini@dit.unitn.it

Fausto Giunchiglia
Department of Information and Communication Technology
University of Trento - Italy - fausto@dit.unitn.it

John Mylopoulos
Department of Computer Science - University of Toronto - Canada -
jm@cs.toronto.edu

Anna Perini
ITC-Irst - Povo (Trento) - Italy - perini@irst.itc.it

March 14, 2002

Abstract.

Our goal in this paper is to introduce and motivate a methodology, called *Tropos*,[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.

Keywords: Agent-Oriented Software Engineering, Multi-Agent Systems, and Agent-Oriented Methodologies

1. Introduction

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

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


© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

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 [28, 30] 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 [30] 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* [20]. 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 mentioned in the first paragraph. 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.

We are defining a software development methodology, called *Tropos*, which will allow us to exploit all the flexibility provided by AOP. In a nutshell, the two keys 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 [25].

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 (see Section 6 for a detailed discussion). 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

[12, 33], and in particular by Eric Yu with his i^* model. Applied in various application areas, including requirements engineering [32], business process reengineering [36], and software modeling processes [35], the i^* model offers actors, goals and actor dependencies as primitive concepts [33]. 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 [1, 22], 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.

Our goal in this paper is to introduce and motivate the Tropos methodology, in all its phases.

The paper is structured as follows. Section 2 introduces the core concepts of the Tropos methodology and provides an early glimpse of how the methodology works. The methodology is then described in Section 3, as applied in the context of the *eCulture system* example, a fragment of a web-based broker of cultural information and services developed for the government of Trentino (Provincia Autonoma di Trento, or PAT). The Tropos modeling language and the diagrammatic representation of the model are introduced in a intuitive way, while a more precise definition of the development process is given in Section 4. The description of the metamodel of the specification language is given in Section 5. A discussion of the related work is presented in Section 6, and finally, the directions of the future work are given in Section 7.

2. The methodology

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.

2.1. 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.

2.2. 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 [21], in Tropos

[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.

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 [34].

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 [7], 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.

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 5.

2.3. 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 5 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 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 but also Section 5 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 9 for an example) and AUML interaction diagrams [22] (Figure 11) are used to this purpose (more details in Section 5).

3. 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 [16]. 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.

3.1. 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 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.

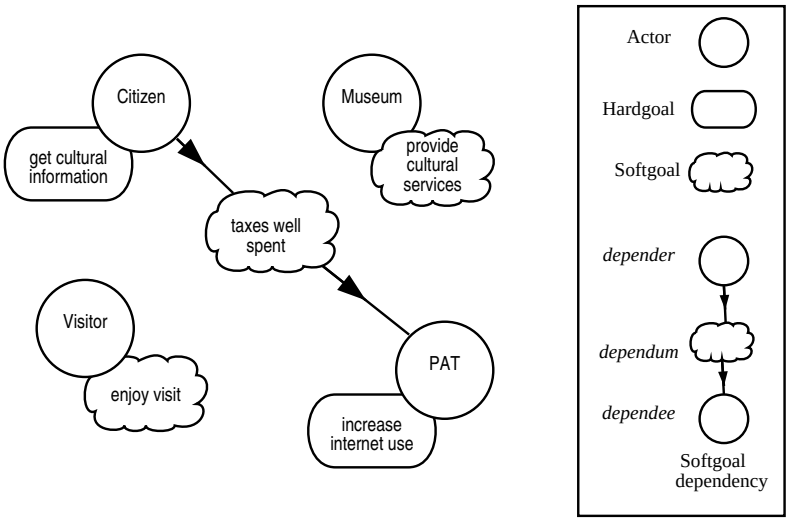

Figure 1. Actor diagram modeling the stakeholders of the eCultural project.

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 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.

A second example, in Figure 3, shows portions of the goal analysis for PAT, relatively to the goals that Citizen delegates to PAT as a result

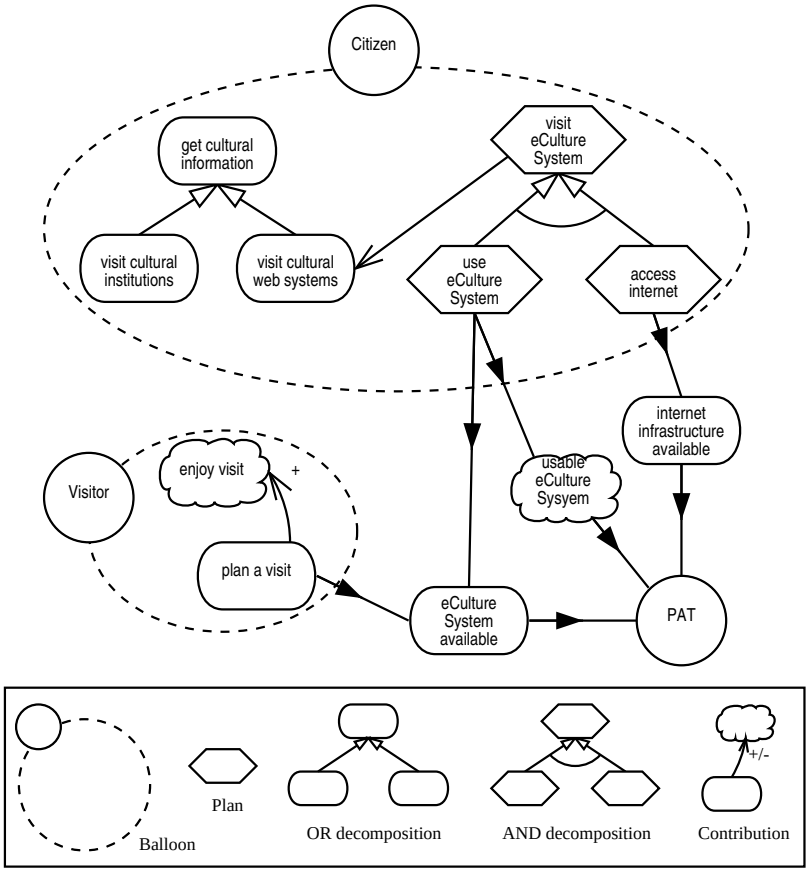

Figure 2. Goal diagrams for Citizen and Visitor. Notice the goal and plan decomposition, the means-end analysis and the (positive)softgoal contribution.

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.

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, the goal build eCulture System is introduced last and, therefore, it has no sub-goals and it is motivated by the higher level goals.

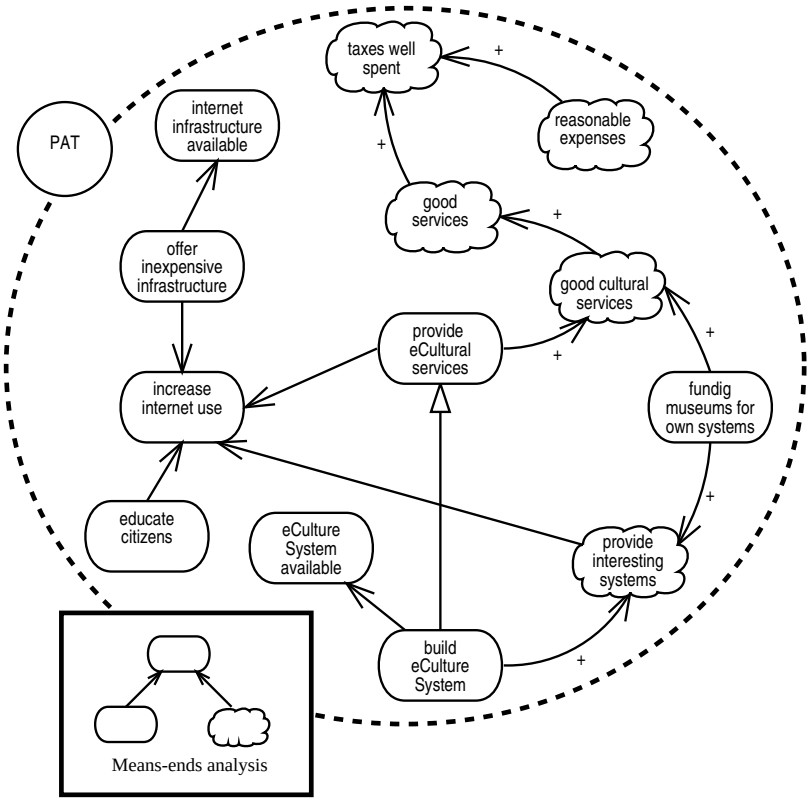

Figure 3. Goal diagram for PAT.

3.2. 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 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), 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 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

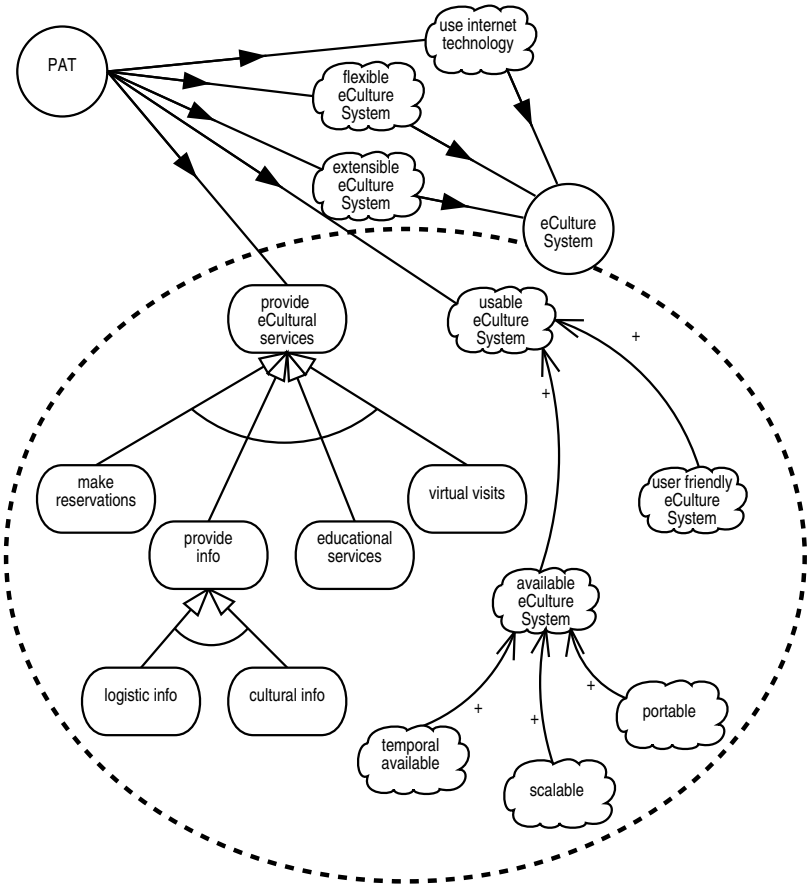

Figure 4. A portion of the actor diagram including PAT and eCulture System and goal diagram of the eCulture System.

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. 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 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 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 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.

3.3. 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;

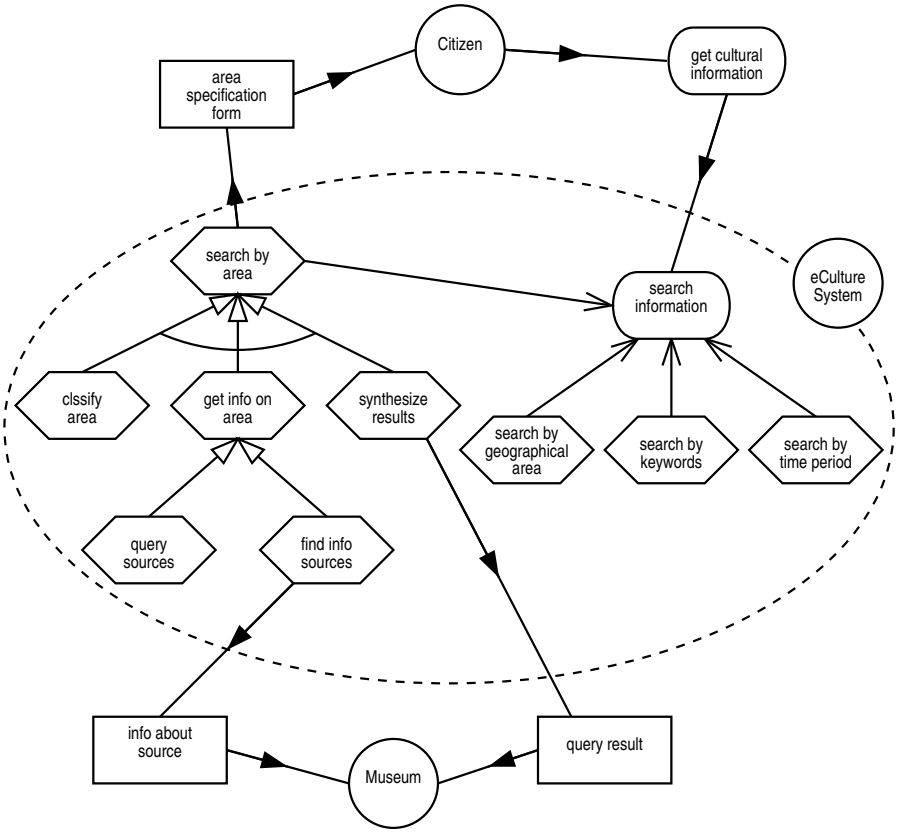

Figure 5. Goal diagram for the goal get cultural information and dependencies between the actor eCulture System and other environment' actors.

- inclusion of new actors according to the choice of a specific architectural style (see [14, 18] for more details about the use of architectural patterns and styles);

- inclusion of actors contributing positively to the fulfillment of some non-functional requirements.

Figure 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.

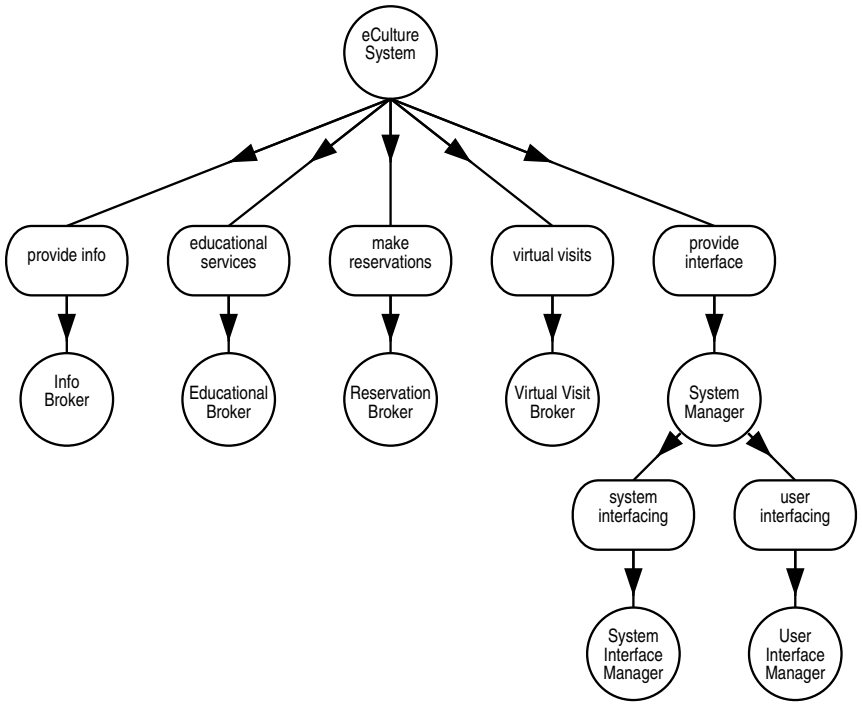

Figure 6. Actor diagram for the eCulture System architecture (step 1).

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 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 7) have been introduced upon the analysis of the plan search by area reported in Figure 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

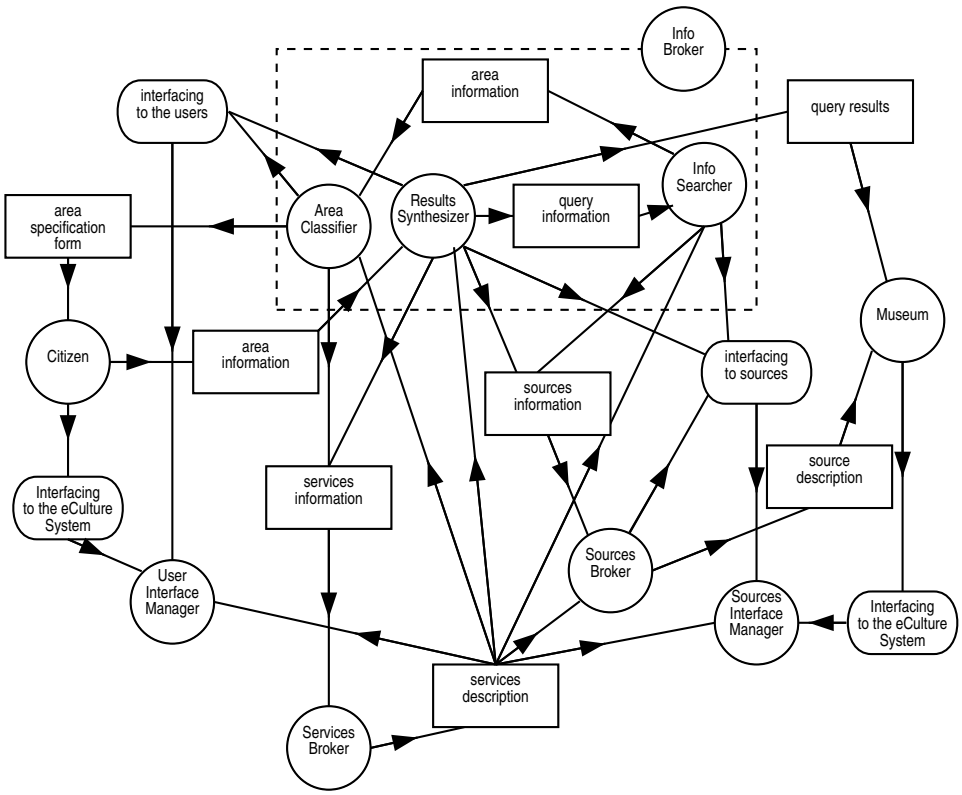

Figure 7. Extended actor diagram w.r.t. the Info Broker (step 1).

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.

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 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

Table I. Actors' capabilities (step 2).

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

Classifier's capabilities as well as the capabilities of the other actors of the extended actor diagram of Figure 7 are listed in Table I.

Step 3. The last step consists of defining a set of agent types and assigning each of them one or more different capabilities (agent as-

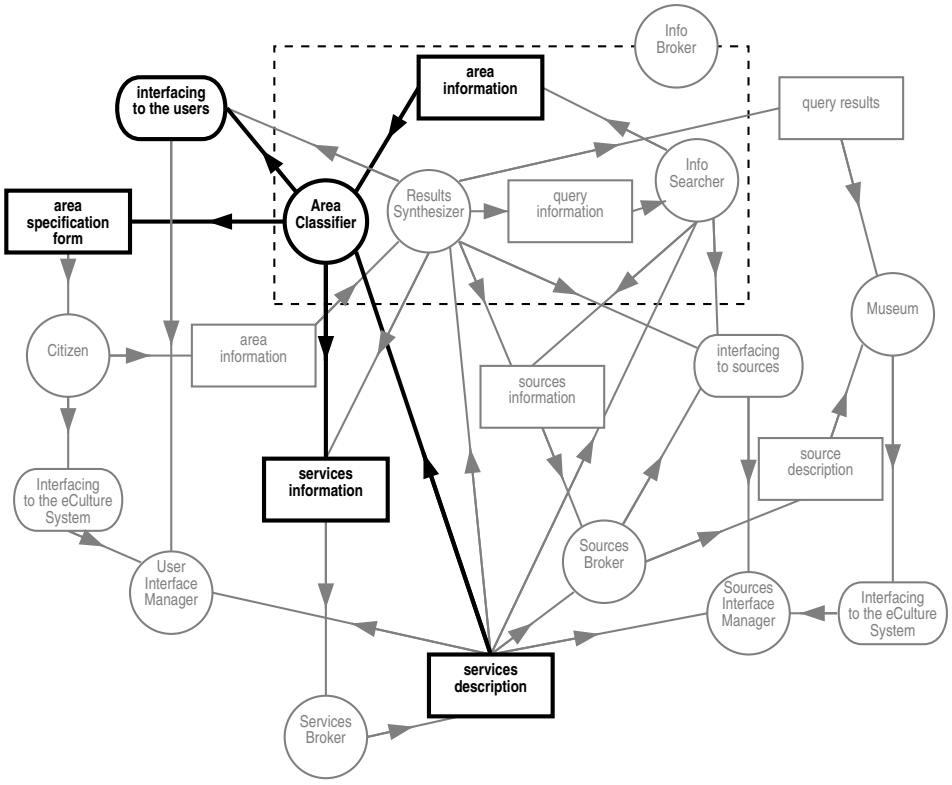

Figure 8. Identifying actor capabilities from actor dependencies w.r.t. the Area Classifier (step 2).

signment). Table II reports the agents assignment with respect to the capabilities identified in Table I. 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.

Table II. Agent types and their capabilities.

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

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 [18].

3.4. Detailed design

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) [2] as the language which should enable the specification of agent systems [1]. Agent UML packages modelling well-known agent communication protocols, such as the Contract Net, are already available [22].

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 [22] 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 9 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 10 depicts the plan evaluate query results belonging to the the capability depicted in the diagram of Figure 9. The plan evaluate query results is activated by the arrival of the query results from the Synthesizer, and it ends storing

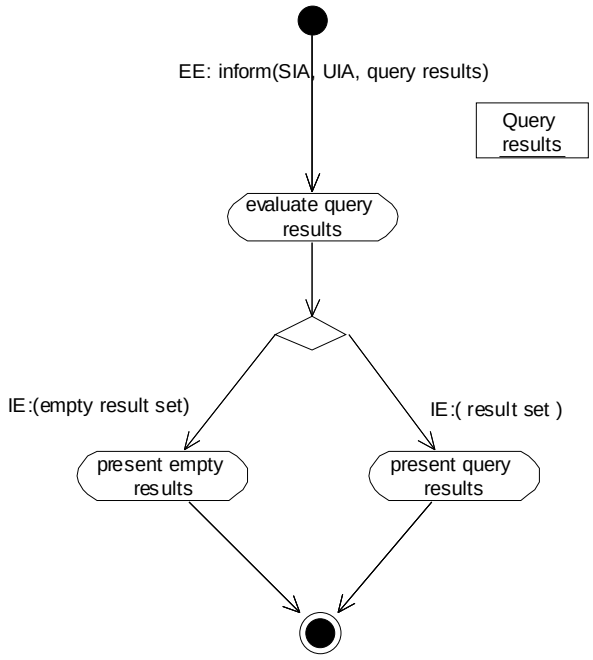

Figure 9. Capability diagram represented as an AUML activity diagram.

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.

A*gent 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 11 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

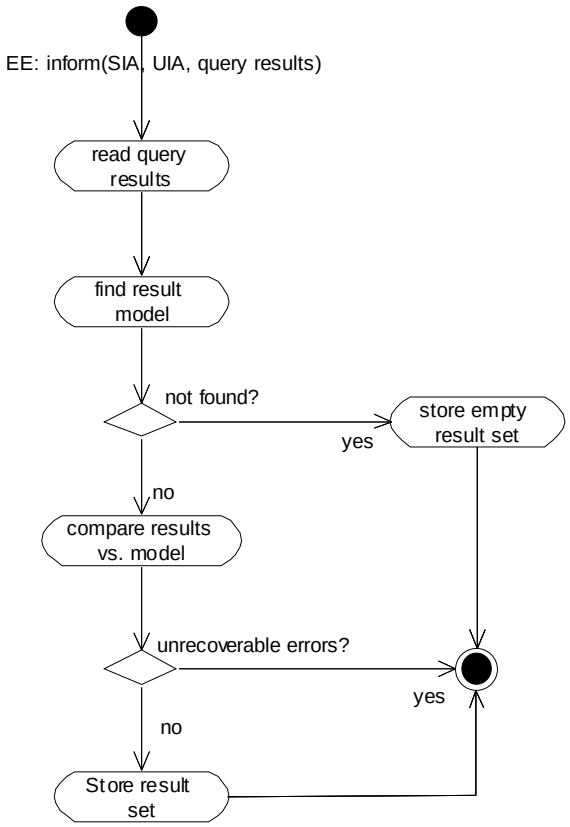

Figure 10. Plan diagram for the plan evaluate query. Ovals correspond to simple or complex actions, arcs to transitions from an action to the subsequent one, start and end states transitions to events.

(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 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 [22] for modeling Agent Interaction

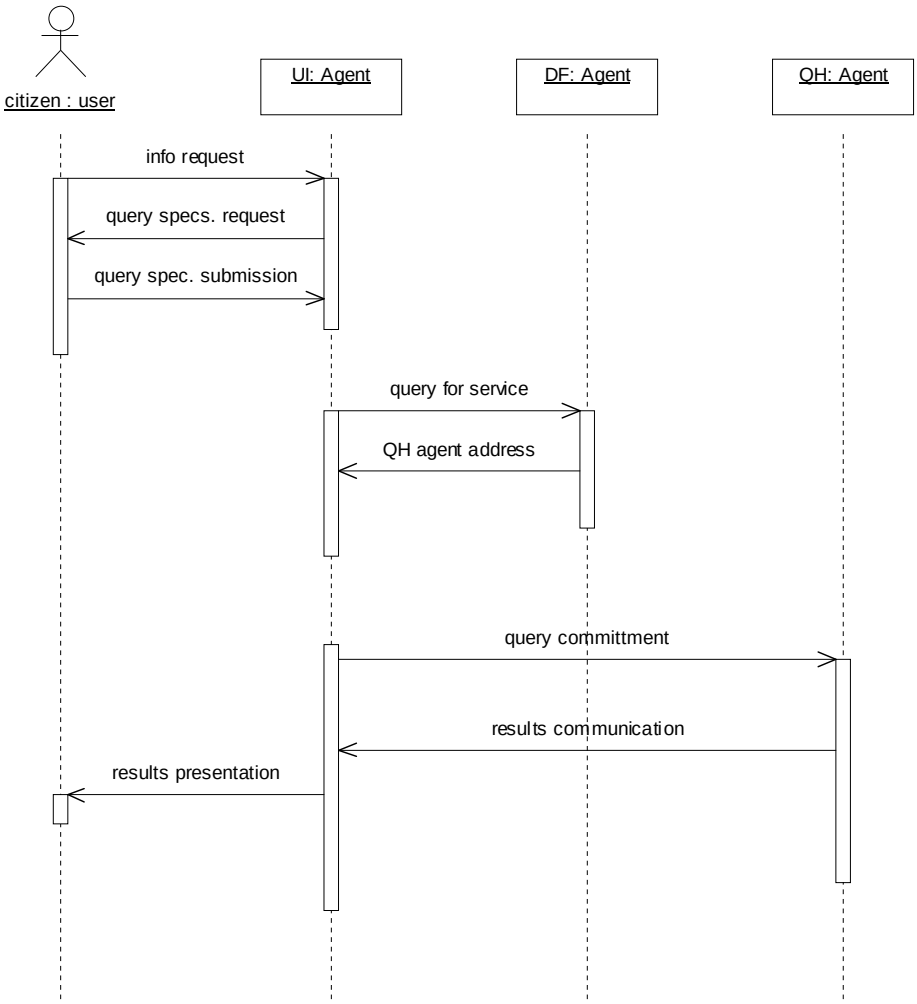

Figure 11. Agent interaction diagram. Boxes represent agents and arrows model communicative acts.

Protocols, can be straightforwardly applied to our example. In such a case, each communicative act of Figure 11 must be analyzed in detail.

3.5. Implementation Using JACK

The BDI platform chosen for the implementation is JACK Intelligent Agents [11], 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 3.4. 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 12 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; }
```

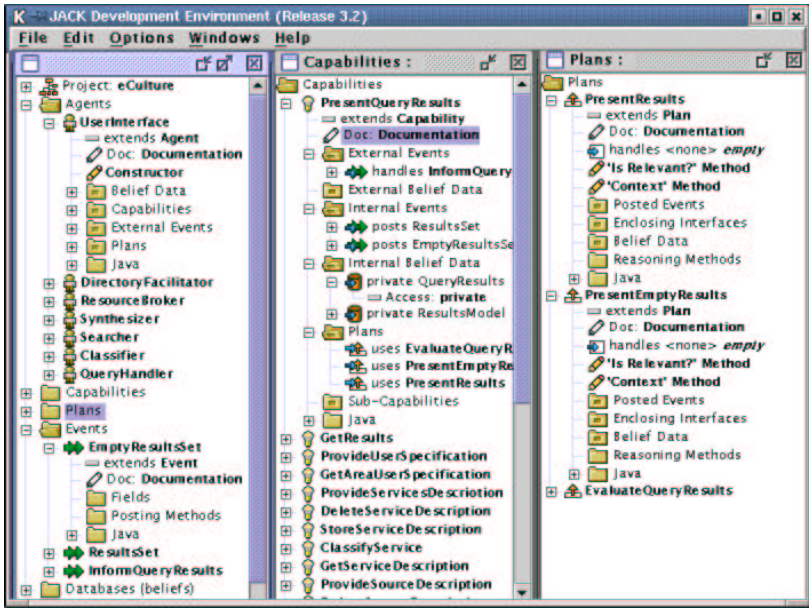

Figure 12. JACK Developing Environment for the eCulture project.

The second window lists all the capabilities associated to the agents of the system. The capability present query results, analyzed in Figure 9, is defined as follows:

```
public capability PresentQueryResults  extends Capability {
    #handles external event InformQueryResults;
    #posts event ResultsSet ;
    #posts event EmptyResultsSet ;
    #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 10), 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); }}}
```

4. 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).

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 $\langle$`goal`, `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 [19]. Its details are beyond the scope of this paper. `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 [15]. 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 [15, 9].

Table III. Tropos language metamodel. The four level architecture.

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

5. 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 [23], the metamodel has been organized in four levels, as shown in Table III. 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 [10]) is handled in [15] and will not be discussed here.

The *Meta-Metamodel* level provides the basis for metamodel extensions. In particular, the meta-metamodel contains language primitives that allows for the inclusions of constructs such as those proposed in [15]. 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 2 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

[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 [27].

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 1 and in Figure 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.

AUML 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.

5.1. The concept of Actor

A portion of the Tropos metamodel concerning the concept of actor is shown in the UML class diagram of Figure 13. *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 1, 4, and 6. The early requirements model depicted in Figure 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

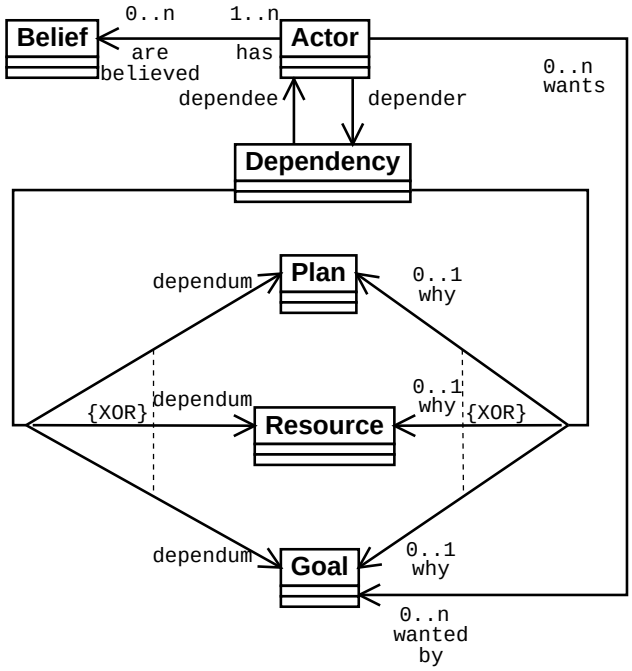

Figure 13. The UML class diagram specifying the actor concept in the Tropos metamodel.

5.2. The concept of Goal

The concept of goal is represented by the class *Goal* in the UML class diagram depicted in Figure 14. 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 a weak form of analysis, consisting of a discovery of goals, plans or resources that can provide means for reaching a goal. Means-end analysis is used in the model shown in Figure 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 14). A contribution can be annotated with a qualitative metric, as used in [7], 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. 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*.

5.3. The concept of Plan

The concept of plan in Tropos is specified by the class diagram depicted in Figure 15. *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.

6. Related Work

As stated in the introduction and also presented in [6], the most important feature of the Tropos methodology is that it aspires to span the overall software development process, from early requirements to implementation. This is represented in Figure 16 which shows the relative coverage of Tropos as well as *i** [33], KAOS [12], GAIA [31], AAII [17] and MaSE [13], and AUML [22, 1, 5]. Many other agent oriented software methodologies have been proposed in the past, see for instance [8, 29, 3, 26]. The considerations applied to the methodologies depicted in Figure 16 apply to these latter methodologies as well.

While Tropos covers the full range of software development phases, it is at the same time well-integrated with other existing work. Thus, for early and late requirements analysis, it takes advantage of work done

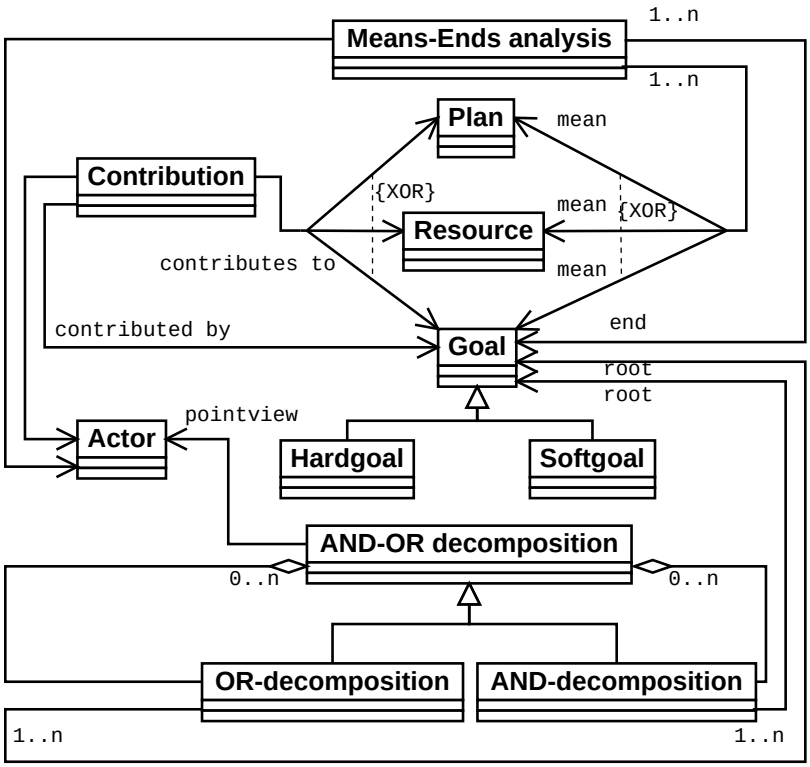

Figure 14. The UML class diagram specifying the the goal concept in the Tropos metamodel.

in the Requirements Engineering community, and in particular of Eric Yu's *i** methodology [33]. As already mentioned above, it is interesting to note that much of the Tropos methodology can be combined with non-agent (e.g., object-oriented or imperative) software development paradigms. For example, one may want to use Tropos for early development phases and then use UML [2] for later phases. At the same time, work on AUML [22] allows us to exploit existing UML techniques during (our version of) agent-oriented software development. As indicated in Figure 16, our idea is to adopt AUML for the detailed design phase. An example of how this can be done is given in [24].

The metamodel presented in Section 5 has been developed in the same spirit as the UML metamodel for class diagrams. A comparison between UML class diagrams and the diagrams presented in Section 5 emphasizes the distinct representational and ontological levels used for class diagrams and actor diagrams (the former being at the software level, the latter at the knowledge level). This contrast also defines the

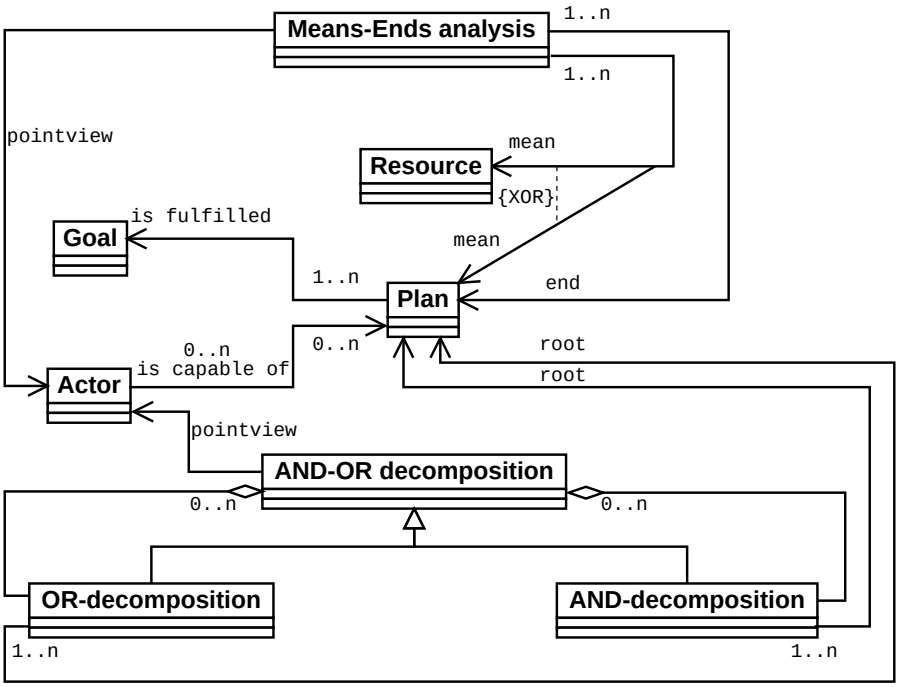

Figure 15. The UML class diagram specifying the plan concept in the Tropos metamodel.

key difference between object-oriented and agent-oriented development methodologies. Agents (and actor diagrams) cannot be thought as a specialization of objects (and class diagrams), as argued in previous papers. The difference is rather the result of an ontological and representational shift. Finally, it should be noted that inheritance, a crucial notion for UML diagrams, plays no role in actor diagrams. This isn't yet a final decision. However inheritance, at the current state of the art, seems most useful at a software, rather than a knowledge, level. This view is implicit in our decision to adopt AUML for the detailed design phase.

7. Conclusions and future work

This paper provides a detailed account of Tropos, a new agent oriented software development methodology which spans the software development process from early requirements to implementation for agent oriented software. The paper presents and discusses the five phases supported by Tropos, the development process within each phase, the

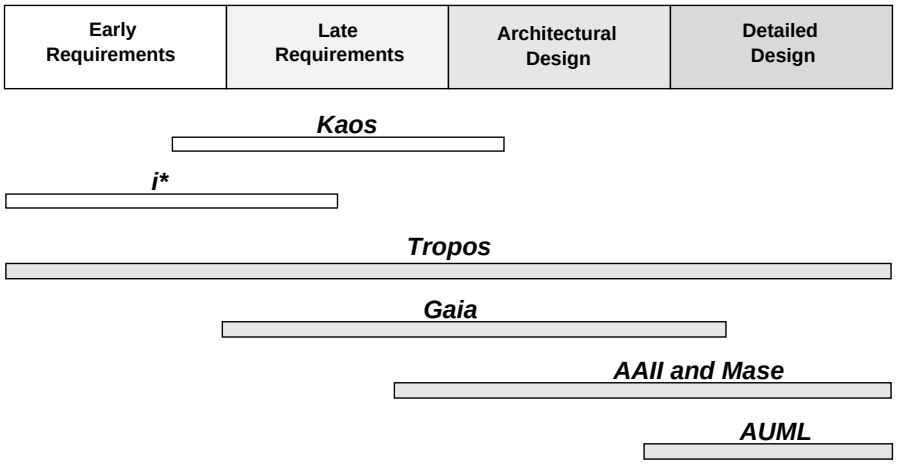

Figure 16. Comparison of Tropos with other software development methodologies.

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 [15]; the formalization of the transformation process in terms of primitive transformations and refinement strategies [4]; the definition of a catalogue of architectural styles for multi-agent systems which adopt concepts from organization theory and strategic alliances literature [18]; 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.

Acknowledgements

We thank all the Tropos Project people working in Trento and in Toronto for useful comments, discussions and feedback.

References

1. 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.
2. G. Booch, J. Rambaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.
3. F.M.T. Brazier, B. Dunin Keplicz, N. Jennings, and J. Treur. DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, 9(1), 1997.
4. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: a transformation based approach. In Wooldridge et al. [29].
5. G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In Wooldridge et al. [29].
6. J. Castro, M. Kolp, and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*. Elsevier, Amsterdam, the Netherlands, (to appear).
7. L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
8. P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*. Springer-Verlag, March 2001.
9. 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.

10. 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.
11. 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.
12. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.
13. S. A. Deloach. Analysis and Design using MaSE and agentTool. In *12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, Miami University, Oxford, Ohio, March 31 - April 1 2001.
14. 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.
15. 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.
16. 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.
17. D. Kinny, M. Georgeff, and A. Rao. A Methodology and Modelling Technique for Systems of BDI Agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Springer-Verlag: Berlin, Germany, 1996.
18. 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.
19. J. Mylopoulos, L. K. Chung, and B. A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, June 1992.
20. A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.
21. H. Nwana. Software agents: An overview. *Knowledge Engineering Review Journal*, 11(3), November 1996.
22. 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.
23. OMG. *OMG Unified Modeling Language Specification*, version 1.3, alpha edition, January 1999.
24. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. In *Proc. of the 5th Int. Conference on Autonomous Agents*, Montreal CA, May 2001. ACM.

25. 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.
26. J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. Using Multi-Context Systems to Engineer Executable Agents. In N. R. Jennings and L. Lesperance, editors, *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, number 1757 in LNCS, pages 277–294. Springer-Verlag, 1999.
27. F. Sannicolò, A. Perini, and F. Giunchiglia. The Tropos modeling language. a User Guide. Technical report, ITC-irst, December 2001.
28. G. Weiss, editor. *Multiagent System: a modern approach to Distributed AI*. MIT Press, 1999.
29. 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.
30. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.
31. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.
32. 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.
33. E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.
34. E. Yu. Agent-oriented modeling: Software versus the world. In Wooldridge et al. [29].
35. 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.
36. 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.