Tropos: An Agent-Oriented Software Development Methodology

PAOLO BRESCIANI AND ANNA PERINI   {bresciani,perini}@irst.itc.it
ITC-Irst, Povo (Trento), Italy

PAOLO GIORGINI AND FAUSTO GIUNCHIGLIA   {paolo.giorgini,fausto}@dit.unitn.it
Department of Information and Communication Technology, University of Trento, Italy

JOHN MYLOPOULOS   jm@cs.toronto.edu
Department of Computer Science, University of Toronto, Canada

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 related mentalistic notions (for instance goals and plans) are used in all phases of software development, from 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.

1. Introduction

Agent Oriented Programming (AOP, from now on) is most often motivated by the need for 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 widely quoted in literature [31, 33] are electronic commerce, enterprise resource planning, air-traffic control systems, personal digital assistants, and so on.

To qualify 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 required [33] are mobility, veracity, rationality, and thee like. The key that makes a software system possess these properties is that it is conceived and programmed at a *knowledge level* [23]. Thus, in AOP, we talk of mental states and beliefs instead of machine states, of plans and actions instead of procedures and methods, of communication, negotiation and social ability instead of interaction and I/O functionalities, of goals, desires, and so on. Explicit representations of such mental notions provide, at least in part, the software with the extra

flexibility needed in order to deal with the intrinsic complexity of applications such as those mentioned earlier. The explicit representation and manipulation of goals and plans facilitates, for instance, a run-time adaptation of system behavior 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 allows us to exploit all the flexibility provided by AOP. In a nutshell, the two novel features of Tropos are:

1. The notion of agent and related mentalistic notions are used in all software development phases, from early requirements analysis down to the actual implementation. Our mentalistic notions are founded on belief, desire, and intention (BDI) agent architectures [28].
2. A crucial role is given to early requirements analysis that precedes the prescriptive requirements specification of the system-to-be. This means that we include in our methodology earlier phases of the software development process than those supported by other agent or object oriented software engineering methodologies (see Section 6 for a detailed discussion). We consider this move as crucial in order to achieve our objectives.

The idea of focusing the activities that precede the specification of software requirements, in order to understand how the intended system will meet organizational goals, is not new. It has been first proposed in requirements engineering, see for instance [12, 36], and specifically in Eric Yu's work with his *i** model. This model has been applied in various application areas, including requirements engineering [35], business process reengineering [39], and software process modeling [38]. The *i** model offers actors, goals and actor dependencies as primitive concepts [36]. The main motivation underlying this earlier work was to develop a richer conceptual framework for modeling processes which involve multiple participants (both humans and software systems). The rationale of the *i** model 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 system dependencies and encourages a uniform treatment of the system's functional and non-functional requirements.

Neither Yu's work, nor, as far as we know, any earlier work on requirements analysis was developed with AOP in mind. The application of these ideas to AOP, and the decision to use mentalistic notions in all phases of analysis, has important consequences. While developing agent oriented specifications and programs, one uses the same notions and abstractions used to describe the behavior of human or social 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 application complexities.

Indeed, the software engineering methodologies and specification languages developed for Object-Oriented Programming (OOP) do not support earlier phases of requirements analysis. This means that there is no formal account or analysis of the

connection between the intentions of the different stakeholders (human, social or otherwise) and the system-to-be. By using UML, for instance, the software engineer can start with use case analysis (possibly refined with activity diagrams) and then move to architectural design. In this phase, the engineer can do static analysis using class diagrams, or dynamic analysis using, for instance, sequence or interaction diagrams. The target is to reach in detail of abstraction level of the actual classes, methods and attributes used to implement the system. However, while applying this approach and related techniques to AOP, the software engineer misses most of the advantages coming for the fact that in AOP one conceives of programs at the knowledge level. UML forces the programmer to translate goals and other mentalistic notions into software level notions, for instance classes, attributes and methods of class diagrams. Consequently, the former notions must be reintroduced in the programming phase. The work on AUML [9, 10] is an example of work suffering from this kind of problem.

The objective of this paper is to introduce and motivate the Tropos methodology, in all its phases. Consistency checking for Tropos models is discussed in [15, 16]. In addition, [6] presents a complementary case study of the Tropos methodology.

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 to *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 its diagrammatic representation are introduced first, 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 related work is presented in Section 6, while Section 7 summarizes the results of the paper and offers directions for future work.

2. The methodology

The Tropos methodology is intended to support all analysis and design activities in the software development process, from 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 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*. The last four phases are well-established in the Software Engineering literature and are supported by various methodologies and tools. The first one (early requirements analysis) is well accepted in the Requirements Engineering research community, but not widely practiced. We then define the basic notions to be modeled during each one of these phases and the techniques that guide model refinement. Finally, we describe the modeling activities performed during the five phases pointing out how the modeling focus shifts with the process.

2.1. Development phases

Requirements analysis represents the initial phase in many software engineering methodologies. As with other approaches, the ultimate objective of requirement analysis in Tropos is to provide a set of functional and non-functional requirements for the system-to-be.

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 initial 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 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* phases 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 sub-systems, interconnected through data and control flows. Sub-systems 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 specific capabilities. The *Detailed Design* phase aims at specifying 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.[2]

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, social or 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 [24], 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 [37].

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. For simplicity, In the rest of the paper goals refer to hard goals when there is no danger of confusion.

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 satisfying 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 actor knowledge of the world.

These notions are more formally specified syntactically 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 sub-systems (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 be-
tween 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 posi-
tively 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
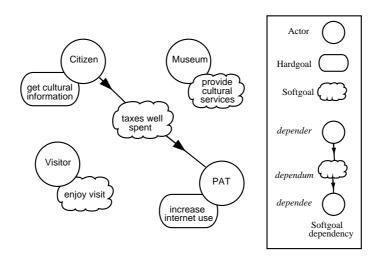

Figure 1. Actor diagram modeling the stakeholders of the eCultural project.

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 [25] (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 [17]. 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.

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 relative 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. It is important to stress that what goals are associated with each actor is a decision of the corresponding stakeholder, not the design team.

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 sub-goals 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 subplans, namely use eCultureSystem 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: plan a visit can give a positive contribution to the goal enjoy visit, and for this the Visitor needs the eCulture System too.
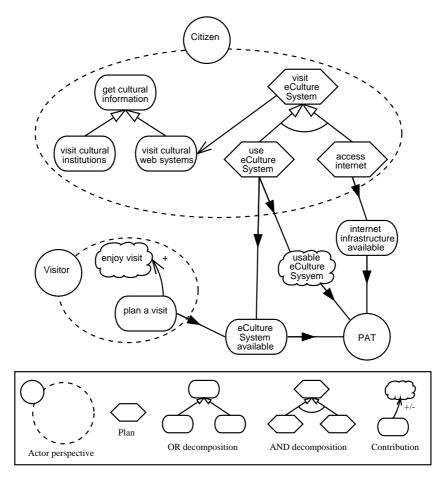
Figure 2. Goal diagrams for Citizen and Visitor. Notice the goal and plan decomposition, the means-end analysis and the (positive)softgoal contribution.

A second example, in Figure 3, shows portions of the goal analysis for PAT, relative 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 soft-goal. The soft-goal 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. Goals lower down in a goal hierarchy are more specific, and are motivated by goals higher up in the hierarchy. For instance, in the example in Figure 3, the goal build eCulture System is motivated by its two supergoals.
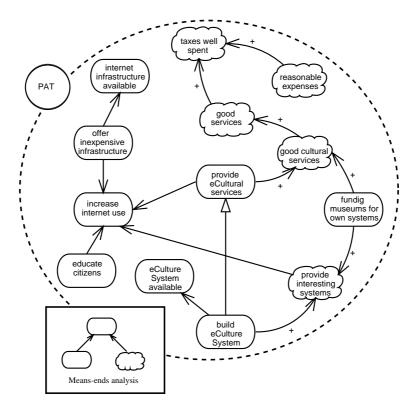
Figure 3. Goal diagram for PAT.

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 sub-goals: 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,
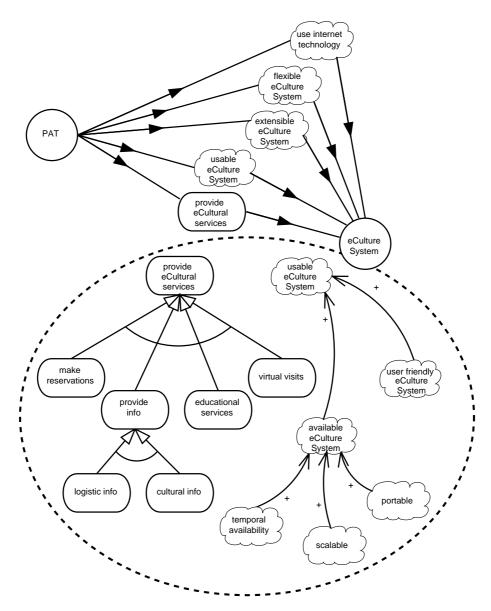
Figure 4. A portion of the actor diagram including PAT and eCulture System and goal diagram of the eCulture System.

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 availability).
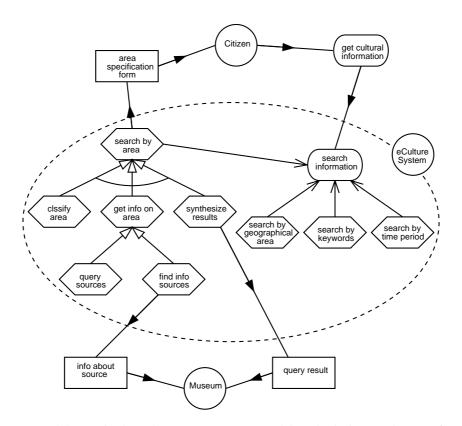
Often, some dependencies in the actor diagram must be revised upon the introduction of the system actor. We have seen in Figure 2 that for Citizen a possible subplan of getting eCultural info is using an eCulture system. Now we can model this in terms of a direct dependency between the actors Citizen and eCulture System. Figure 5 shows how this dependency is analyzed inside the goal diagram of the eCulture System. The goal search information (a sub-goal 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 querysources, 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.

The analysis conducted so far is intended to provide a context within which the system-to-be is to be designed. Skipping this analysis can lead to misunderstandings about what the system should be doing or the special qualities it should possess. Indeed, it has been well documented in the Software Engineering literature that many software faults and failures originate in misunderstood requirements [18].

3.3. Architectural design

The architectural design phase defines the system's global architecture in terms of sub-systems (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

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 and delegation of sub-goals to sub-actors upon goal analysis of system's goals,
– inclusion of new actors according to the choice of a specific architectural style (see [14, 21] for more details about the use of architectural patterns and styles),
– inclusion of actors contributing positively to the fulfillment of some specific functional and non-functional requirement.

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.

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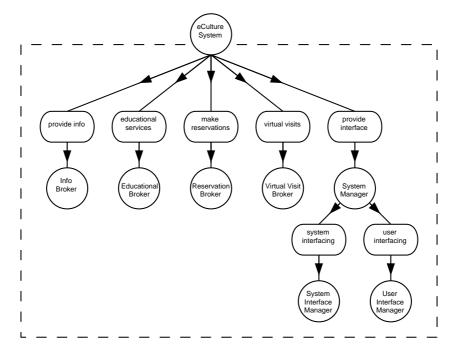
Figure 6. Actor diagram for the eCulture System architecture (step 1).

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 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 are not derived automatically but they can be easily identified by analyzing the extended actor diagram. In particular,
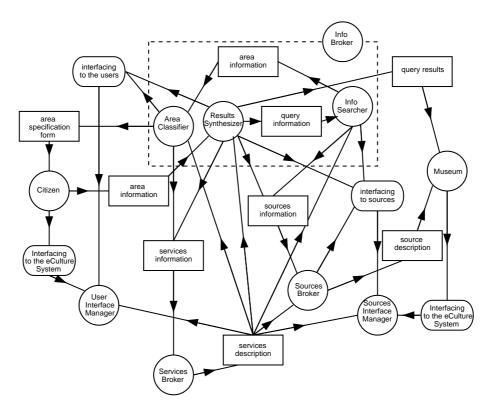
Figure 7. Extended actor diagram w.r.t. the Info Broker (step 1).

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 Classifier's capabilities as well as the capabilities of the other actors of the extended actor diagram of Figure 7 are listed in Table 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 2 reports the agents assignment with respect to the capabilities identified in Table 1. Of course, many other capabilities and agent types are needed in case we consider all the goals and plans associated with the complete extended actor diagram.

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 [21].
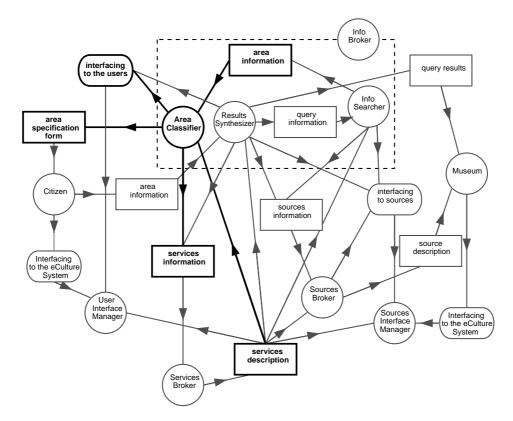
Figure 8. Identifying actor capabilities from actor dependencies w.r.t. the Area Classifier (step 2).

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 Foundation for Intelligent Physical Agents (FIPA) is 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 modeling well-known agent communication protocols, such as the Contract Net, are already available [25].

In Tropos, we adapt existing results from these approaches to agent system design. However, our detailed design step takes as input 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 capabilities and plans, and we adopt a subset of the AUML diagrams proposed in [25] for specifying agent protocols.

Table 1. 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

Capability 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; action states model plans, transition arcs model finishing of action states, 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.

Plan diagrams.   Each plan node of a capability diagram can be further specified by UML activity diagrams. For instance, Figure 10 depicts the plan evaluate results corresponding to the capability depicted in Figure 9. The plan evaluate query results

Table 2. 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

is activated by the arrival of the query results from the Synthesizer, and it ends storing an empty or non-empty result set. Query results are compared to a set of possible result models contained in an agent's beliefs. Possible errors during the comparison end the plan without any side effect. 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 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
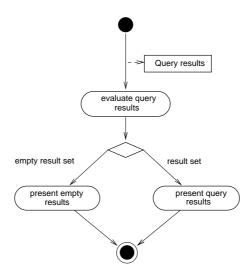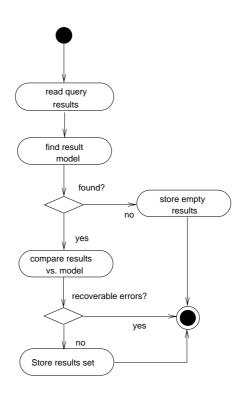

Figure 9. Capability diagram represented as an UML activity diagram.

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.

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 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 [25] for modeling Agent Interaction 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
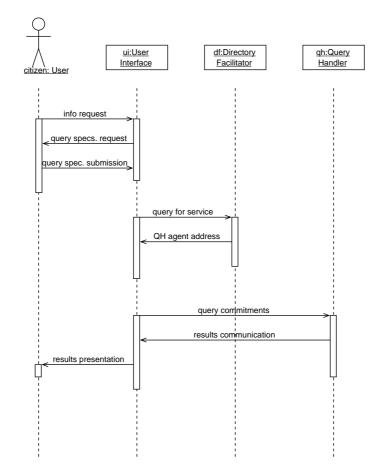
Figure 11. Agent interaction diagram. Boxes represent agents and arrows model communicative acts.

in order to make them capable of achieving goals. The implementation activity follows 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 agent 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 capability 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 notion provides a means to reuse.
– *Belief*: The JACK database amounts to a generalized relational database that describes a set of beliefs ascribed to an agent.
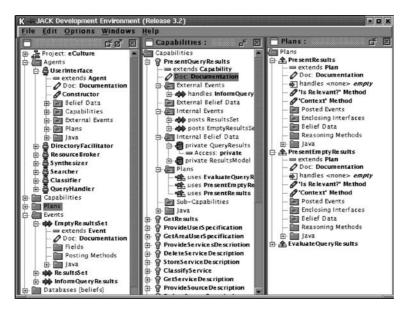
Figure 12. JACK Developing Environment for the eCulture project.

– *Event*: Internal and external events specified in the detailed design map to JACK's event construct. In JACK, an event describes a triggering condition for agents actions.
– *Plan*: The plans contained in a capability specification resulting from a detailed design map to JACK plans. In JACK, a plan is a sequence of instructions the agent follows to try to achieve goals and deal with occurrences of 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 UserlnterfaceextendsAgent {
    #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 with 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 with 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 [19]. 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 sub-goals 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 ⟨goal, plan⟩ 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, goal Graph 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 [22]. 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 sub-goal 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 sub-goals 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 four non-deterministic operations in `goalAnalysis` are highlighted with times-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 `goal Analysis`, for the lack of space, others are left to the imagination of the reader. `Delegate Goal` 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 sub-goals that have been generated from this analysis. During architectural design, more system actors are introduced and are delegated sub-goals 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 and 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 [11, 27].

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 [26]. Tropos has exploited standard four-layer metadata

Table 3. 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

architecture, as shown in Table 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 [10]) is handled in [15] 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 [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,[3] let us present the Tropos model and diagrams.

A Tropos model is a directed labeled graph whose nodes are instances of meta-classes 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 Figures 1 and 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
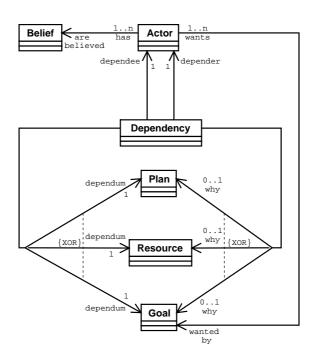
Figure 13. The UML class diagram specifying the actor concept in the Tropos metamodel.

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 (labeled *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 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).

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 sub-classes *Hardgoal* and *Softgoal*, respectively.

Goals can be analyzed, from the point of view of an actor, performing *means-end analysis*, *contribution* 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, 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 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 labeled *contributes to* in Figure 14). A contribution can be annotated with a qualitative metric, as used in [7], aenoted by $+, ++, -, --$. In particular, if the goal $g1$ contributes positively to the goal $g2$, with metric $++$ then if $g1$ is
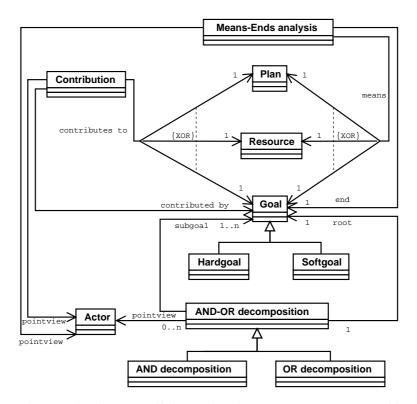

Figure 14. The UML class diagram specifying the the goal concept in the Tropos metamodel.

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 sub-goals. The particular case where the root goal $g1$ is decomposed into a single sub-goal $g2$, 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^* [36], KAOS [12], GAIA [34], AAII [20] and MaSE [13], and AUML [1, 5, 25]. Many other agent oriented software methodologies have been proposed in the past, see for instance [3, 8, 29, 32]. The considerations raised for the methodologies shown 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 in the Requirements Engineering community, and in particular Eric Yu's i^* methodology [36]. As already noted, much of the Tropos methodology can be combined with non-agent (e.g., object-oriented or imperative) software development techniques. 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 [25] allows us to exploit existing AUML techniques adapted for 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 [27].
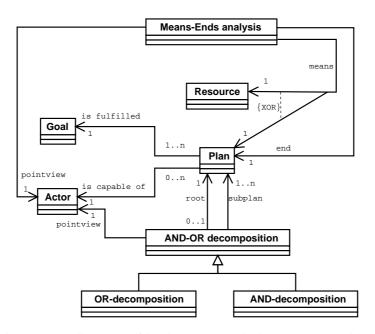
Figure 15. The UML class diagram specifying the plan concept in the Tropos metamodel.
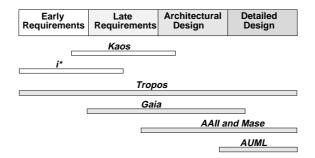

Figure 16. Comparison of Tropos with other software development methodologies.

The metamodel presented in Section 5 has been developed in the same spirit as the UML metamodel for class diagrams. A comparison between class diagrams meta-model and the metamodels for 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 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 is not yet a final decision. However

inheritance, at the current state of the art seems more 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 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.

Of course, the Tropos methodology is not intended for any type of software. For system software (such as a compiler) or embedded software, the operating environment of the system-to-be is an engineering artifact, with no identifiable stakeholders. In such cases, traditional software development techniques may be most appropriate. However, a large and growing percentage of software does operate within open, dynamic organizational environments. For such software, the Tropos methodology and others in the same family apply and promise to deliver more robust, reliable and usable software systems. The Tropos methodology in its current form is also not suitable for sophisticated software agents requiring advanced reasoning mechanisms for plans, goals and negotiations. Further extensions will be required to the Tropos methodology, mostly at in detailed design phase, to address this class of software applications.

Our long term objective is to provide a 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 problems, 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 [22]; the extension of the coverage of the development process by addressing some aspects currently not fully considered, as requirements elicitation, at the very beginning of the process, and testing, deployment and maintenance; and, finally, the development of tools which support the methodology, at least during some 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 (although not all of them) 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.

Acknowledgments

Notes

1. From the Greek "tropé", which means "easily changeable", also "easily adaptable".
2. Notice that Tropos (as well as other agent-oriented software engineering methodologies) can be used independently of the fact that one uses AOP as implementation technology.
3. 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 [30].

References

1. B. Bauer, J. P. Müller, and J. Odell, "Agent UML: A formalism for specifying multiagent software systems," *Int. J. Software. Eng. Knowl. Eng.*, vol. 11, no. 3, pp. 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," *Int. J. Coop. Inform. Syst.*, vol. 9, no. 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. [17].
5. G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massoneta, "Agent oriented analysis using MESSAGE/UML," in Wooldridge, et al. [17].
6. J. Castro, M. Kolp, and J. Mylopoulos. "Towards requirements-driven information systems engineering: The tropos project," *Information Systems*, Elsevier: Amsterdam, The Netherlands.
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, (ed.), *Agent-Oriented Software Engineering, vol. 1957 of Lecture Notes in AI.*, Springer-Verlag, 2001.

9. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *Int. J. Software Tools Technol. Transf.* (*STTT*), vol. 2, no. 4, 2000.

10. E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in D. Kozen, (ed.), *Proceedings of the Workshop on Logics of Programs, vol. 131 of Lecture Notes in Computer Science*, Yorktown Heights, New York, Springer-Verlag: New York, pp. 52–71, 1981.

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. Fickasu, "Goal-directed requirements acquisition," *Sci. Comput. Program.*, vol. 20, no. 1–2, pp. 3–50, 1993.

13. S. A. Deloach, "Analysis and design using MaSE and agent Tool, 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 *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, Toronto, CA, August 2001.

16. P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with goal models," in S. Spaccapietra, S. T. March, and Y. Kambayashi, (eds.), *21st International Conference on Conceptual Model.* (*ER02*), *Tampere, Finland, vol. 2503 of Lecture Notes in Computer Science*, Springer-Verlag, 2002.

17. 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, (eds.), *Proceedings of the Thirteenth International Conference on Software Engineering – Knowledge Engineering* (*SEKE01*), Buenos Aires, ARGENTINA, June 13–15, 2001.

18. A. Davis and F. Giunchiglia, *Software Requirements*: *Objects, Functions and States*, Prentice Hall, 1993.

19. F. Giunchiglia, J. Mylopoulos, and A, Perini, "The Tropos software development methodology: Processes, models and diagrams," in F. Giunchiglia, J. Odell, and G. Weiß, (eds.), *Agent-Oriented Software Engineering III, Third International Workshop* (*AOSE2002*), *Bologna, Italy* LNCS, Springer-Verlag, in press.

20. 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, (eds.), *Agents Breaking Away*: *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Springer-Verlag: Berlin, Germany, 1996.

21. M. Kolp, P. Giorgini, and J. Mylopoulos, "An goal-based organizational perspective on multi-agents architectures," in *Procedings of the 8th International Workshop on Agent Theories, Architectures, and Languages* (*ATAL-2001*), Seattle, WA, August 2001.

22. J. Mylopoulos, L. K. Chung, and B. A. Nixon, "Representing and using non functional requirements: A process-oriented approach," *IEEE Trans. Software Eng.*, 1992.

23. A. Newell, "The knowledge level," *Artif. Intell.*, vol. 18, pp. 87–127, 1982.

24. H. Nwana, "Software agents: An overview," *Knowl. Eng. Rev. J.*, vol. 11, no. 3, 1996.

25. J. Odell, H. Parunak, and B. Bauer, "Extending UML for agents," in G. Wagner, Y. Lesperance, and E. Yu, (eds.), *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, TX, 2000, pp. 3–17.

26. OMG, *OMG Unified Modeling Language Specification*, version 1.3, Alpha edition, January 1999.

27. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos, "A knowledge level software engineering methodology for agent oriented programming," in *Proceedings of the 5th Int. Conference on Autonomous Agents*, Montreal CA, May 2001, ACM.

28. 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.

29. J. Sabater, C. Sierra, S. Parsons, and N. R. Jenning, "Using multi-context systems to engineer executable agent," in N. R. Jennings and L. Lesperance, (eds), *Proceedings of the 6th International*

Workshop on Agent Theories Architectures, and Languages (*ATAL-99*), number 1757 in LNCS, Springer-Verlag, 1999, pp. 277–294.

30. F. Sannicol'91o, A. Perini, and F. Giunchiglia, "The Tropos modeling language. A User Guide," Technical report, ITC-irst, December 2001.

31. G. Weiss, (ed.), *Multiagent System*: *A modern approach to Distributed AI*, MIT Press, 1999.

32. M. Wooldridge, P. Ciancarini, and G. Weiss (eds.), *Proceedings of the 2nd International Workshop on Agent-Oriented Software Engineering* (*AOSE-2001*), Montreal, CA, May 2001.

33. M. Wooldridge and N. R. Jeanings, "Intelligent agents: Theory and practice," *Knowl. Eng. Rev.*, vol. 10, no. 2, 1995.

34. M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia methodology for agent-oriented analysis and design," *J. Autonomous Agents Multi-Agent Sys.*, vol. 3, no. 3, 2000.

35. E. Yu, "Modeling organizations for information systems requirements engineering," in *Proceedings of the First IEEE International Symposium on Requirements Engineering*, San Jose, January lEEE, 1993, pp. 34–41.

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

37. E. Yu, "Agent-oriented modeling: Software versus the world," in Wooldridge et al. [17].

38. 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.

39. E. Yu and J. Mylopoulus, "Using goals, rules, and methods to support reasoning in business process reengineering," *Int. J. Intell. Syst. Account. Finance Manage.*, vol. 1, no. 5, 1996.