Towards an Agent Oriented approach to Software Engineering

Anna Perini and Paolo Bresciani
ITC-IRST
Via Sommarive 18, 38055 Povo, Trento, Italy
{perini,bresciani}@irst.itc.it

Paolo Giorgini and Fausto Giunchiglia
Dep. of Information and Communication Tech.
University of Trento
via Sommarive 14, Povo, Trento, Italy
{pgiorgini,fausto}@ict.unitn.it

John Mylopoulos
Department of Computer Science
University of Toronto
M5S 3H5, Toronto, Ontario, Canada
jm@toronto.edu

Abstract

This paper describes a methodology for agent oriented software engineering, called Tropos[1]. *Tropos is based on three 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. Third, Tropos adopts a transformational approach to process artifacts refinement. The methodology is partially illustrated with the help of a case study.*

1. Introduction

Advanced software applications call most often 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 recompilation, and with minimal assumptions about its operating environment and its users. One of the most critical dimension of complexity of this type of software is communication between components. In other words, this type of software applications require to deal with aspects that traditionally have been ascribed to multi-agents systems. Similar considerations can be pursued at the system specification level. These intuitions motivates recent efforts in adapting concepts and methodologies for Agent Oriented Programming (AOP) to the development of complex software systems, in analogy with what happened when concepts of Object Oriented Programming (OOP) were discovered to be useful for the analysis and design of software systems, independently of the use of OOP as implementation technology.

To qualify as an agent, a software or hardware system is often required to have properties such as autonomy, social ability, reactivity, proactivity. Other attributes which are sometimes requested are mobility, veracity, rationality, and so on. The key feature that makes it possible to implement systems with the above properties is that, in this paradigm, programming is done at a very abstract level, more precisely, following Newell, at the *knowledge level* [12]. Thus, in agent oriented programming, we talk 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. Abstract mental notions are essential in order to provide, at least in part, the software with the extra flexibility needed to deal with the complexity intrinsic in the mentioned applications. The explicit representation and manipulation of goals and plans allows, for instance, for a runtime "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.

Agent oriented programming is often introduced as a specialization or as a "natural development" of object oriented programming, see for instance [16, 10, 17]. In our opinion, the step from object oriented programming to agent oriented programming is more a paradigm shift than

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

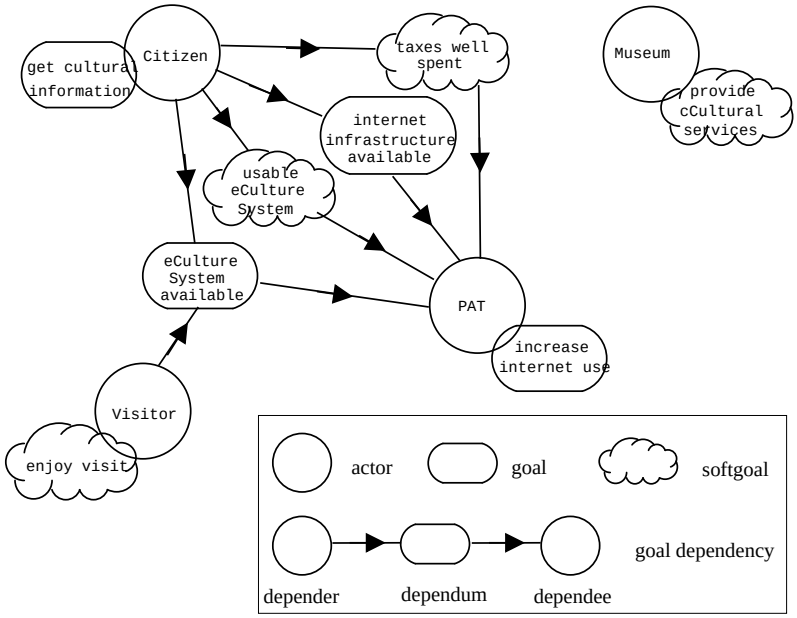

Figure 1. An actor diagram specifying the project stakeholders and their main goal dependencies.

a simple specialization. Also those features of agent oriented programming which can be found in object oriented programming languages, for instance, mobility and inheritance, take, in our context, a different and more abstract meaning.

Several approaches to agent-oriented software engineering have been developed, ranging from structured, informal methodologies, to formal ones, as described in a recent overview [1] and in [2] , most of them focusing basically on architectural design.

We propose a software development methodology, called *Tropos* [14], which will allow us to exploit all the flexibility provided by agent oriented programming. In a nutshell, the three key features of Tropos are the following:

1. The notion of agent and all the related mentalistic notions are used in all phases of software development, from the first phases of early analysis down to the actual implementation.

2. A crucial role is given to the earlier analysis of requirements that precedes prescriptive requirements specification. We consider therefore much earlier phases with respect to standard object oriented methodologies as, for instance, those based on the Unified Modeling Language (UML) [6], where use case analysis is proposed as an early activity, followed by architectural design.

3. The methodology rests on the idea of building a model of the system-to-be that is incrementally refined and extended from a conceptual level to executable artifacts. This process adopts a transformational approach: a set of transformation operators which allow the engineer to progressively detail the higher level notions introduced in the earlier phases are proposed. It must be noticed that, contrarily to what happens in most other approaches, e.g., UML based methodologies, there is no change of graphical notation from one step to the next (e.g., from use cases to class diagrams). The refinement process is performed in a more uniform way.

In the following section we give an overview of the Tropos methodology, partially illustrated with examples extracted from a case-study described in [14]. Some conclusions are presented in Section 3.

2. The Tropos Methodology

The Tropos methodology is intended to support all the analysis and design activities from the very early phases of requirements engineering down to implementation, and organizes them into five main development phases: early requirement analysis, late requirements analysis, architectural design, detailed design and implementation[2].

The Tropos modeling language is derived from the Eric Yu's i* paradigm [18] which offers actors, goals, and actor dependencies as primitive concepts for modeling an application during early requirements analysis. Tropos' lan-

[2]The concept of phase in Tropos denotes a set of activities of the software development process with a logical coherence. Elsewhere this concept is denoted as workflow, for instance in the Rational Unified Process (RUP)[5]

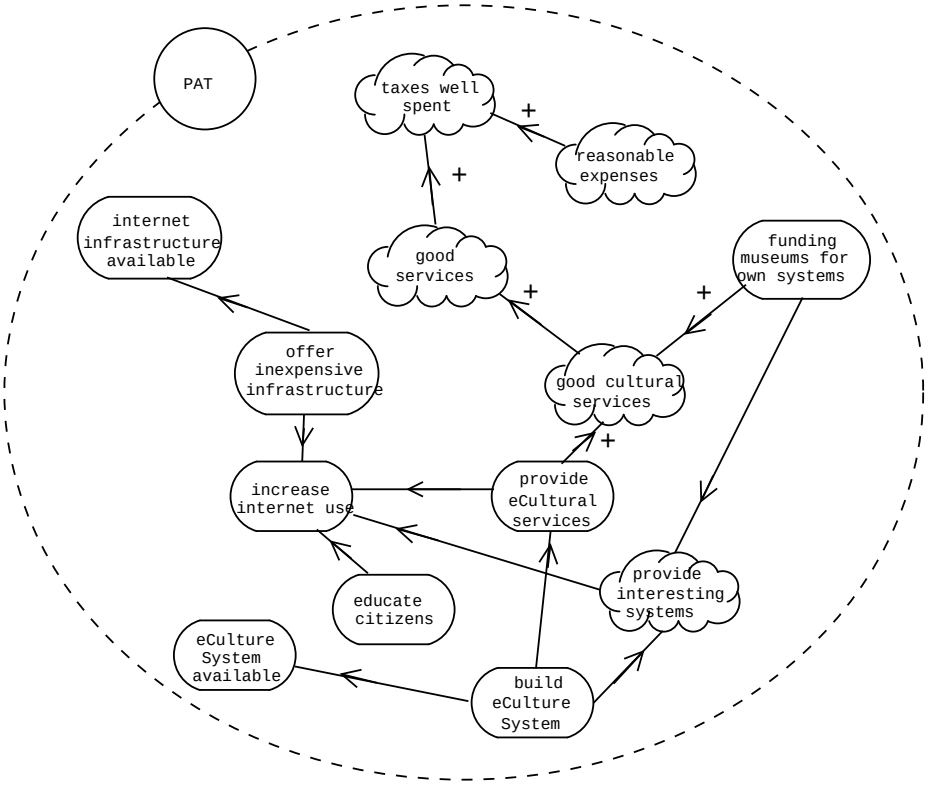

Figure 2. A goal diagram for PAT. The analysis shows goal decomposition and softgoal (positive) contribution.

guage is intended to support both an informal model specification and a formal one, allowing for automatic checking of model properties [15]. A set of diagrammatic representation of the model are provided. Each element in the model has its own graphical representation, taken from the i* framework. Two main types of diagrams are provided for visualizing the model: the *actor diagram*[3], where the nodes (the actors) are connected through dependencies (labeled arcs), and the *goal diagram*[4], represented as a balloon labeled with a specific actor name and containing goal and plan analysis, conducted from the point of view of the actor.

In the rest of this section we briefly describe the five Tropos phases, specifying the main activities of each phase and the process artifacts. The example are extracted from a case-study which refers to the development of a web-based broker of cultural information and services (herein *eCulture system*) for the Province of Trentino, including information obtained from museums, exhibitions, and other cultural organizations. It is the government's intention that the system be usable by a variety of users, including citizens of Trentino and tourists looking for things to do, or scholars and students looking for material relevant to their studies.

[3]See the *i** strategic dependencies diagram.

[4]See the *i** rationale diagram.

2.1. Early Requirements

The main objective of the early requirement analysis in Tropos is the understanding of a problem by studying an existing organizational setting. During this phase, the requirement engineer models the stakeholders as actors and analyzes their intentions, that are modeled as goals which, through a goal-oriented analysis, are then decomposed into finer goals, that eventually can support evaluation of alternatives. Goal analysis can be concluded by identifying plans that, if performed by the actor, allow for goal achievement. The analysis can also lead to the identification of further dependencies with other actors. When necessary, we distinguish between hard and soft goals, the latter lacking a clear-cut definition and/or criteria for deciding whether they are satisfied or not. Softgoals are amenable to a more qualitative kind of analysis that, when moving to later phases concerning the system definition, may lead to the identification of non-functional requirements. The resulting model can be depicted as an actor diagram.

Figure 1 shows the actors involved in the eCulture

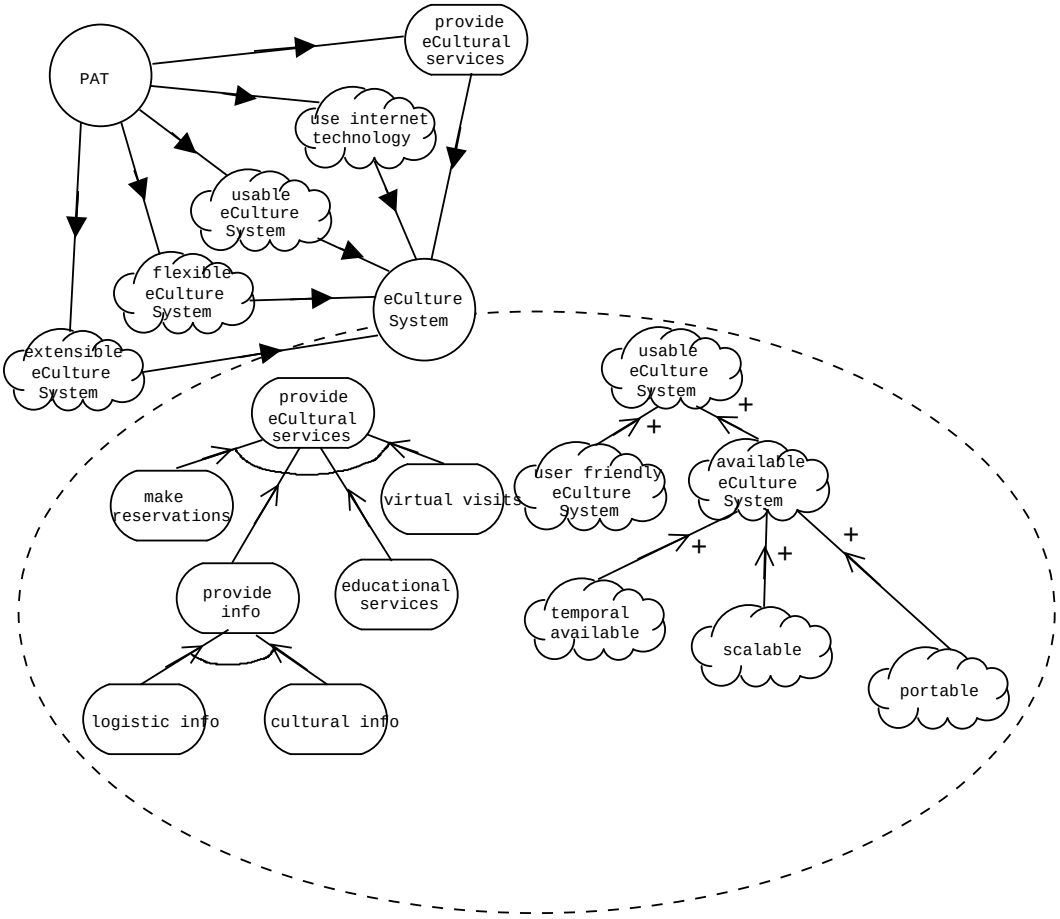

Figure 3. A fragment of the actor diagram including the `PAT` and the `eCulture System` and the goal diagram for the `eCulture System`.

project and their respective goals. In particular, the actor `PAT` represents the local government and has been represented with a single relevant goal: `increase internet use`. The actors `Visitor` and `Museum` have associated softgoals, `enjoy visit` and `provide cultural services` respectively. The actor `Citizen` wants to get cultural information and depends on `PAT` to fulfill the softgoal `taxes well spent`, a high level goal that motivates more specific `PAT`'s responsibilities, namely to provide an Internet infrastructure, to deliver on the `eCulture system` and make it usable too. Some of the dependencies in Figure 1 arise from a refinement of the preliminary model obtained by performing goal analysis, as depicted, for instance, in Figure 2.

2.2. Late Requirements

The late requirement analysis aims at specifying the system-to-be within its operating environment, along with relevant functions and qualities. The system is represented as an actor which have a number of dependencies with the actors already described during the early requirements phase. These dependencies define all functional and non-functional requirements for the system-to-be. The actor diagram in Figure 3 includes the `eCulture System` and shows a set of goals that `PAT` delegates to it through goal dependencies. These goals are then analyzed from the point of view of the `eCulture System` and are shown in the goal diagram depicted in the lower part of Figure 3. In the example we concentrate on the analysis for the goal `provide eCultural services` and the softgoal `usable eCulture System`. The goal `provide eCultural services` is decomposed (AND decomposition) into four subgoals: `make reservations`, `provide info`, `educational services` and `virtual visits`. As basic eCultural service, the `eCulture System` must provide information (`provide info`), which can be `lo-`

gistic info, and cultural info. Softgoal contributions are than identified. 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).

2.3. Architectural Design

The main objective of the architectural design phase is the definition of the system's global architecture in terms of subsystems (actors), interconnected through data and control flows (dependencies). Basically, this phase consists of three steps: refining the system actor diagram introducing subactors upon analysis of functional and non functional requirements and taking into account design patterns (step 1); capturing actor capabilities from the analysis of the tasks that actors and sub-actors will carry on in order to fulfill functional requirements (step 2); defining a set of agent types (components) and in assigning to each component one or more different capabilities (step 3). A portion of the architectural design model of the eCulture project, resulting from the first step, is represented by the actor diagram in Figure 4.

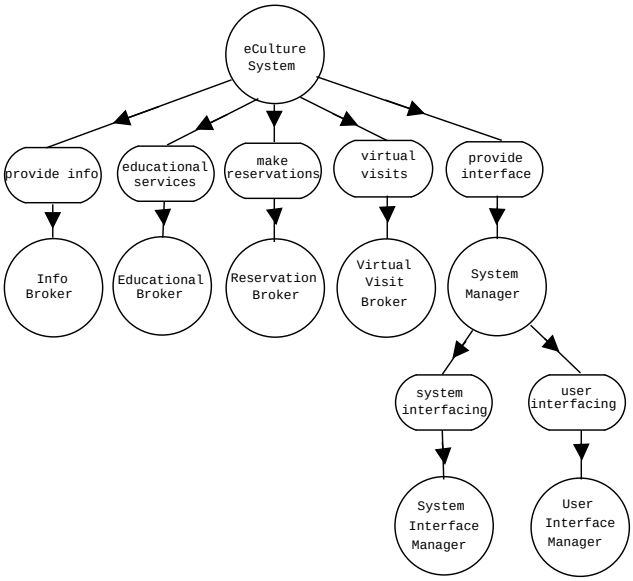

Figure 4. Actor diagram of the architecture of the eCulture System (step 1)

2.4. Detailed Design

The detailed design phase aims at specifying the agent (component) 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[5]. So, for instance, choosing a BDI (Belief Desire Intention) multi-agent platform will require the specification of agent capabilities in terms of external and internal events that trigger plans, and the beliefs involved in agent reasoning. These properties are specified through a set of diagrams. A subset of the AUML diagrams proposed in [3, 13] are used: the activity diagrams (*capability diagram*) to model a capability (or a set of correlated capabilities) from the point of view of a specific actor; activity diagrams (*plan diagram*) to specify each plan node of a capability diagram; and sequence diagrams (*agent interaction diagram*) to model agents interaction in terms of communication acts.

2.5. Implementation

The implementation activity follows step by step the detailed design specification, according to the established mapping between the implementation platform constructs and the detailed design notions. In our case-study, the JACK Intelligent Agents [7] platform has been chosen for implementation. JACK is a BDI agent-oriented development environment built on top and fully integrated with Java, where agents 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.

3. Conclusions

In this paper we have proposed Tropos, a new software engineering methodology which allows us to exploit the advantages and the extra flexibility (if compared with other programming paradigms, for instance OOP) coming from using AOP. Two main intuitions underlying Tropos are the pervasive use, in all phases, of knowledge level specifications, and the idea that one should start from the very early phase of early requirements specification. This allows us to create a continuum where one starts with a set of mentalistic notions (e.g., beliefs, goals, plans), always present in (the *why* of) early requirements, and to progressively converts them into the actual mentalistic notions implemented in an agent oriented software. This direct mapping from the early requirements down to the actual implementation allows us

[5]Note that agent oriented software engineering methodologies are recognized as promising approaches to the development of complex systems [1], independently of the use of AOP as implementation technology.

to develop software architectures which are "well tuned" with the problems they solve and have, therefore, the extra flexibility needed in the complex applications mentioned in the introduction.

Several open points still remain. The most important are: complete the definition of the Tropos language metamodel including useful concepts such as beliefs and events; formalize the transformational approach defining both primitive transformations and refinement strategies[4], provide the methodology with a catalogue of architectural styles for multi-agent systems which adopt concepts from organization theory and strategic alliances literature [11].

References

[1] P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*. Springer-Verlag, 2001.

[2] M. Wooldridge, P. Ciancarini and G. Weiss, organizers. Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada - May 29th 2001.

[3] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In Ciancarini and Wooldridge [1].

[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. [2].

[5] J. Jacobson, G. Booch, and J. Rambaugh. *Unified Software Development Process*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

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

[7] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. AOS Technical Report tr9901, Jan. 1999. http://www.jackagents.com/pdf/tr9901.pdf.

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

[9] A. Dardenne, A. van Lamsweerde, and S. Fickas. "goal" directed requirements acquisition. *Science of Computer Programming*, (20), 1993.

[10] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2), 2000.

[11] M. Kolp, P. Giorgini and J. Mylopoulos. An Goal-Based Organizational Perspective on Multi-Agents Architectures. Eighth International Workshop on Agent Theories, architectures, and languages (ATAL-2001), Seattle, USA, August, 2001.

[12] A. Newell. The knowledge level. *Artificial Intelligence*, 18, 1982.

[13] B. B. James Odell, H. Van Dyke Parunak. Representing agent interaction protocols in UML. In Ciancarini and Wooldridge [1].

[14] A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. Autonomous Agents, Montreal CA, May 2001.

[15] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model Checking Early Requirements Specifications in Tropos. Fifth IEEE International Symposium on Requirements Engineering, Toronto, CA, August 2001.

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

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

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