The Tropos Software Development Methodology: Processes, Models and Diagrams

Fausto Giunchiglia[1], John Mylopoulos[2], and Anna Perini[3]

[1] Department of Information and Communication Technology, University of Trento
via Sommarive, 14, I-38050 Povo, Italy - fausto@dit.unitn.it
[2] Department of Computer Science, University of Toronto
M5S 3H5, Toronto, Ontario, Canada -jm@cs.toronto.edu
[3] ITC-Irst, Via Sommarive, 18, I-38050 Trento-Povo, Italy
phone +39 0461 314 330 - perini@irst.itc.it

Abstract. *Tropos* is a novel agent-oriented software development methodology founded on two key features: *(i)* the notions of agent, goal, plan and various other knowledge level concepts are fundamental primitives used uniformly throughout the software development process; and *(ii)* a crucial role is assigned to requirements analysis and specification when the system-to-be is analyzed with respect to its intended environment. This paper provides a (first) detailed account of the Tropos methodology. In particular, we describe the basic concepts on which Tropos is founded and the types of models one builds out of them. We also specify the analysis process through which design flows from external to system actors through a goal analysis and delegation. In addition, we provide an abstract syntax for Tropos diagrams and other linguistic constructs.

1 Introduction

New application areas such as eBusiness, application service provision and peer-to-peer computing call for software systems which have open, evolving architectures, operate robustly and exploit resources available in their environment. To build such systems, practicing software engineers are discovering the importance of mechanisms for communication, negotiation, and coordination between software components. We expect that many will be turning to multi-agent system technologies and methodologies for guidance and support in building the software systems of the future.

We are developing a comprehensive software engineering methodology, named *Tropos*, for multi-agent systems.

In a nutshell, the two key features of Tropos are: *(i)* the use of knowledge level [9] concepts, such as agent, goal, plan and other through all phases of software development, and *(ii)* a pivotal role assigned to requirements analysis when the environment and the system-to-be is analyzed.

The phases covered by the proposed methodology are as follows.

Early Requirements: during this phase the relevant stakeholders are identified, along with their respective objectives; stakeholders are represented as actors, while their objectives are represented as goals;

Late Requirements: the system-to-be is introduced as another actor and is related to stakeholder actors in terms of actor dependencies; these indicate the obligations of the system towards its environment, also what the system can expect from actors in its environment;
Architectural Design: more system actors are introduced and they are assigned sub-goals or subtasks of the goals and tasks assigned to the system;
Detailed Design: system actors are defined in further detail, including specifications of communication and coordination protocols;
Implementation: during this phase, the Tropos specification, produced during detailed design, is transformed into a skeleton for the implementation. This is done through a mapping from the Tropos constructs to those of an agent programming platform such as JACK [2]; code is added to the skeleton using the programming language supported by the programming platform.

The Tropos methodology has been motivated and illustrated with case studies, see for instance [11, 12]. The purpose of this paper is to present the methodology in further detail. The rest of the paper is structured as follows. Section 2 presents the Tropos primitive knowledge level concepts used for building the different types of models, and illustrates them with examples. Section 3 describes the analysis process that guides model evolution through different development phases. The Tropos modeling language is then defined in Section 4 in terms of UML diagrams, while related work is discussed in Section 5. Conclusions and directions for further research are presented in Section 6.

2 Concepts and Models

The Tropos conceptual models and diagrams are developed as instances of the following intentional and social concepts: actor, goal, dependency, plan, resource, capability, and belief. Below we discuss each one in turn.
Actor. The notion of actor models an entity that has strategic goals and intentionality. An actor represents a *physical agent* (e.g., a person, an animal, a car), or a *software agent* as well as a *role* or a *position*. A *role* is an abstract characterization of the behavior of an actor within some specialized context, while 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. Notice that the notion of actor in Tropos is a generalization of the classical AI notion of software agent.
Goal. A goal represents the strategic interests of actors. Our framework distinguishes between hard goals and softgoals, the latter having no clear-cut definition and/or criteria as to whether they are satisfied. Softgoals are useful for modeling software qualities [3], such as security, performance and maintainability.
Dependency. A dependency between two actors indicates that one actor depends on another 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 (goal, plan resource) around which the dependency centers is called *dependum*. By depending on other actors, 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,

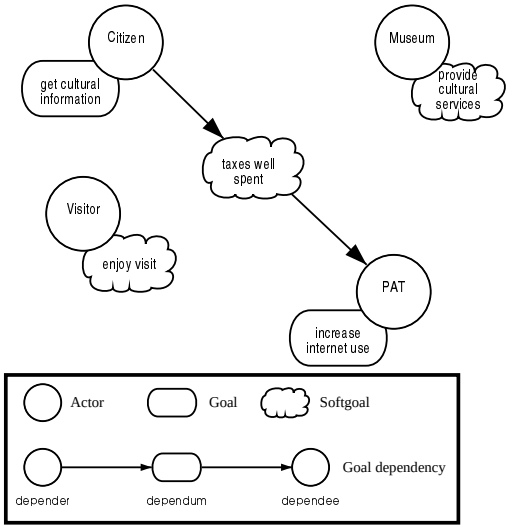

Fig. 1. Actor diagram of the stakeholders of the eCulture System.

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.

Plan. A plan represents a way of satisfying a goal.

Resource. A resource represents a physical or an informational entity that one actor wants and another can deliver.

Capability. A capability represents the ability of an actor to define, choose and execute a plan to fulfill a goal, given a particular operating environment.

Belief. Beliefs are used to represent each actor's knowledge of the world.

Notice how the notions of belief, goal (or desire), and plan (or intention) are the key concepts of the BDI framework. The notion of dependency, instead, is quite interesting and novel, and it turns out to be very important when modeling the intentional inter-dependencies between actors

These concepts can be used to build different types of models throughout the development process. We illustrate these with examples extracted from a substantial software system developed for the government of Trentino (Provincia Autonoma di Trento, or PAT), and partially described in [11]. The system (which we will call throughout the *eCulture system*) is a web-based broker of cultural information and services for the province of Trentino, including information obtained from museums, exhibitions, and other cultural organizations and events to be used by a variety of users, including Trentino citizens and visitors.

We consider, in turn, examples of actor, dependency, goal and plan models. Other types of models are not discussed here for lack of space; see [12] for more examples.

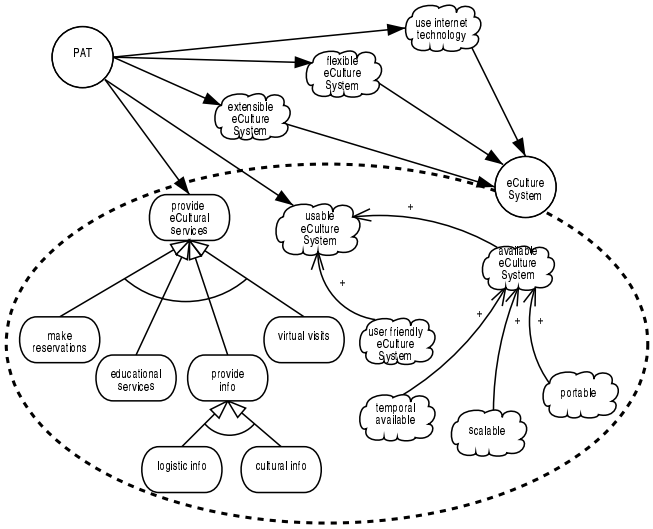

Fig. 2. An actor diagram including PAT and eCulture System and a goal diagram of the eCulture System.

2.1 Actor and Dependency models

Actor and dependency models result from the analysis of social and system actors, as well as of their goals and dependencies for goal achievement. These types of models are built in the early requirements phase when we focus on characterizing the application domain stakeholders, their intentions and the dependencies that interleave them. Actor and dependency models are graphically represented through *actor diagrams* in which actors are depicted as circles, their goals as ovals and their softgoal as cloud shapes. The network of dependency relationships among actors are depicted as two arrowed lines connected by a graphical symbol varying according to the dependum: a goal, a plan or a resource. Figure 1 shows the *actor diagram* for the eCulture domain as resulting from a first early requirement analysis. In particular, the actor Citizen is associated with a single relevant goal: get cultural information, while the actor Visitor has an associated softgoal enjoy visit. Along similar lines, the actor PAT wants to increase internet use for Trentino citizens, while the actor Museum wants to provide cultural services.

Actor models are extended during the late requirements phase by adding the system-to-be as another actor, along with its inter-dependencies with social actors. For example, in Figure 2 the actor PAT delegates a set of goals to the eCulture System actor through goal dependencies namely, provide eCultural services, which is a goal that contributes to the main goal of PAT increase internet use and softgoals such as extensible eCulture System, flexible eCulture System, usable eCulture System, and use internet technology.

Actor models at the architectural design level provide a more detailed account of the system-to-be actor and its internal structure. This structure is specified in terms of subsystem actors, interconnected through data and control flows that are modeled as

dependencies. This model provides the basis for capability modeling, an activity that will start later on during the architectural design phase, along with the mapping of system actors to software agents.

2.2 Goal and Plan models

Goal and plan models allow the designer to analyze goals and plans from the perspective of a specific actor by using three basic reasoning techniques: *means-end analysis*, *contribution analysis*, and *AND/OR decomposition*. For goals, means-end analysis proceeds by refining a goal into subgoals in order to identify plans, resources and softgoals that provide means for achieving the goal (the end). Contribution analysis allows the designer to point out goals that can contribute positively or negatively in reaching the goal being analyzed. In a sense, contribution analysis can be considered as a special case of means-end analysis, where means are always goals. AND/OR decomposition allows for a combination of AND and OR decompositions of a root goal into sub-goals, thereby refining a goal structure.

Goal models are first developed during early requirements using initially-identified actors and their goals. Figure 3, shows portions of the goal model for PAT, relative to the goals that Citizen has delegated to PAT through an earlier goal analysis. Goal and plan models are depicted through *goal diagrams* that represent the perspective of a specific actor as a balloon that contains graphs whose nodes are goals (ovals) and /or plans (hexagonal shape) and whose arcs represents the different types of relationships that can be identified between its nodes. In Figure 3, the goals increase internet use and eCulture System available are both well served (through a contribution relationship) by the goal build eCulture System. Within an actor balloon, softgoal analysis is also performed identifying positive or negative contributions from other goals. The softgoal taxes well spent gets positive contributions from the softgoal good services, and the goal build eCulture System

Goal models play an analogous role in identifying (and justifying) actor dependencies during late requirements and architectural design. Figure 2 shows a goal diagram for the eCulture System, developed during late requirements analysis. In the example we concentrate on the goal provide eCultural services and the softgoal usable eCulture System. The goal provide eCultural services is AND-decomposed into four subgoals make reservations, provide info, educational services and virtual visits. The goal (provide info) is further decomposed into (the provision of) logistic info, concerning timetables and visiting information for museums and cultural info. Virtual visits are services that allow Citizen to pay a virtual visit to a city of the past (e.g., Rome during Cæsar's time!). Educational services include presentation of historical and cultural material at different levels of detail (e.g., at a high school or undergraduate university level) as well as on-line evaluation of the student's grasp of this material. Make reservations allows Citizen to make reservations for particular cultural events, such as concerts, exhibitions, and guided museum visits.

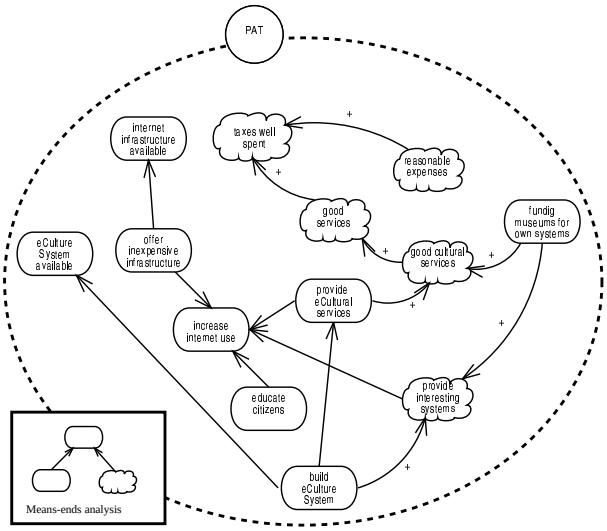

Fig. 3. Goal diagram for PAT.

3 The Development Process

In this section we focus on the generic design process through which these models are constructed. The process is basically one of analyzing goals on behalf of different actors, and is described in terms of a non deterministic concurrent algorithm, including a completeness criterion. Note that this process is carried out by *software engineers* (rather than software agents) at *design-time* (rather than run-time).

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

Assume that *actorList* includes a finite set of actors, also that the list of goals for *actor* is stored in *goalList(actor)*. In addition, we assume that *agenda(actor)* includes the list of goals *actor* has undertaken to achieve personally (with no help from other actors), along with the plan that has been selected for each goal. Initially, *agenda(actor)* is empty. *dependencyList* includes a list of dependencies among actors, while *capabilityList(actor)* includes $< goal, plan >$ 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, *goalGraph* contains all root goals of all initial actors with no links among them. We will treat all of the above as global variables which are accessed and/or updated by the procedures pre-

sented 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 [8]. Its details are beyond the scope of this paper. *rootGoalAnalysis* succeeds if there is a set of non-deterministic selections within the concurrent executions of *goalAnalysis* procedures which leads to the satisfaction of all root goals.

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

During early requirements, this process analyzes initially-identified goals of external actors ("stakeholders"). At some point (late requirements), the system-to-be is introduced as another actor and is delegated some of the subgoals that have been generated from this analysis. During architectural design, more system actors are introduced and are delegated subgoals to system-assigned goals. Apart from generating goals and actors in order to fulfill initially-specified goals of external stakeholders, the development process includes specification steps during each phase which consist of further specifying each node of a model such as those shown in Figure 3. Specifications are given in a formal language (*Formal Tropos*) described in detail in [7]. These specifications add constraints, invariants, pre- and post-conditions which capture more of the semantics of the subject domain.

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

4 The Tropos Modeling Language

The modeling language is at the core of the Tropos methodology. The abstract syntax of the language is defined in this section in terms of a UML metamodel. Following

Table 1. The four level architecture of the Tropos metamodel.

Level	Description	Examples
meta metamodel	Basic language structural elements	Attribute, Entity
metamodel	Knowledge level notions	Actor, Goal, Dependency
domain	Application domain entities	PAT, Citizen, Museum
instance	Domain model instances	Mary: instance of Citizen

standard approaches [10], the metamodel has been organized in four levels, as shown in Table 1. The four-layer architecture makes the Tropos language extensible in the sense that new constructs can be added. Semantics for the language is handled in [7] and won't be discussed here.

The **meta-metamodel** level provides the basis for metamodel extensions. In particular, the meta-metamodel contains language primitives that allows for the inclusions of constructs such as those proposed in [7]. The **metamodel** level provides constructs for modeling knowledge level entities and concepts. The **domain** model 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** model level contains instances of the domain model.

We focus below only the metamodels for goals. [4]

The concept of goal is represented by the class Goal in the UML class diagram depicted in Figure 4. The distinction between hard and softgoals is captured through a specialization of Goal into subclasses Hardgoal and Softgoal respectively.

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

Means-ends analysis is a ternary relationship defined among an Actor, whose point of view is represented in the analysis, a goal (the end), and a Plan, Resource or Goal (the means). Means-end analysis is a weak form of analysis, consisting of a discovery of goals, plans or resources that can provide means for reaching a goal. Means-end analysis is used in the model shown in Figure 3, where the goals educate citizens and provide eCultural services, as well as the softgoal provide interesting systems are means for achieving the goal increase internet use.

Contribution analysis is a ternary relationship between an Actor, whose point of view is represented, and two goals. Contribution analysis strives to identify goals that can contribute positively or negatively towards the fulfillment of a goal (see association re-

[4] 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 [12].

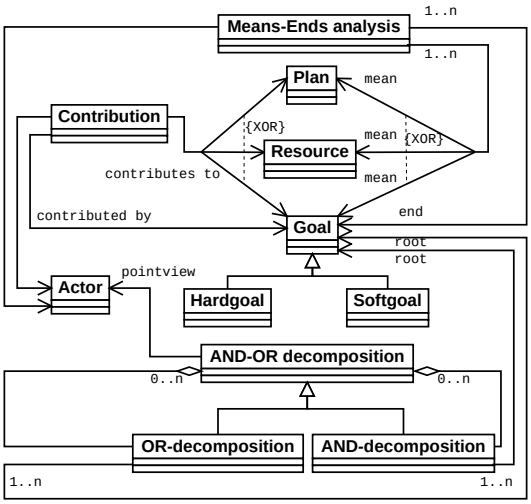

Fig. 4. The goal concept specified by an UML class diagram.

lationship labelled contributes to in Figure 4). A contribution can be annotated with a qualitative metric, as used in [3], denoted by $+, ++, -, --$. In particular, if the goal g1 contributes positively to the goal g2, with metric $++$ then if g1 is satisfied, so is g2. Analogously if the plan p contributes positively to the goal g, with metric $++$, this says that p fulfills g. A $+$ label for a goal or plan contribution represents a partial, positive contribution to the goal being analyzed. With labels $--$, and $-$ we have the dual situation representing a sufficient or partial negative contribution towards the fulfillment of a goal. Examples of contribution analysis are shown in Figure 3. For instance the goal funding museums for own systems contributes positively to both the softgoals provide interesting systems and good cultural services, and the latter softgoal contributes positively to the softgoal good services.
AND-OR decomposition is also a ternary relationship which defines an AND- or OR-decomposition of a root goal into subgoals. The particular case where the root goal g1 is decomposed into a single subgoal g2, is equivalent to a ++ contribution from g2 to g1.

5 Related Work

As indicated in the introduction, 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 5 which shows the relative coverage of Tropos as well as *i** [14], KAOS[5], GAIA [13], AAII [4], MaSE [6], and AUML [1].

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

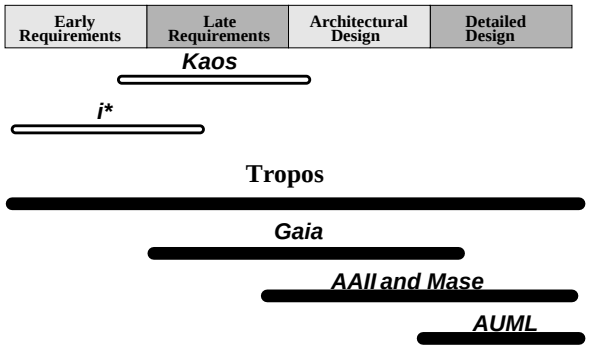

Fig. 5. Comparison of Tropos with other software development methodologies.

analysis, it takes advantage of work done in the Requirements Engineering community, and in particular of Eric Yu's *i** methodology [14]. It is interesting to note that much of the Tropos methodology can be combined with non-agent (e.g., object-oriented or imperative) software development paradigms. For example, one may want to use Tropos for early development phases and then use UMLfor later phases. At the same time, work on AUML [1]allows us to exploit existing UML techniques during (our version of) agent-oriented software development. As indicated in Figure 5, our idea is to adopt AUML for the detailed design phase. An example of how this can be done is given in [11].

The metamodel presented in Section 4 has been developed in the same spirit as the UML metamodel for class diagrams. A comparison between UML class diagrams and the diagrams presented in Section 4 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.

6 Conclusion

This paper provides a detailed account of Tropos, an 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 (in part) 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.

Acknowledgments

We thank all the Tropos Project people working in Trento and in Toronto.

References

1. B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.
2. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical Report TR9901, AOS, January 1999. http://www.jackagents.com/pdf/tr9901.pdf.
3. L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
4. M. Georgeff D. Kinny and A. Rao. A Methodology and Modelling Technique for Systems of BDI Agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Springer-Verlag: Berlin, Germany, 1996.
5. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.
6. S. A. Deloach. Analysis and Design using MaSE and agentTool. In *12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, Miami University, Oxford, Ohio, March 31 - April 1 2001.
7. A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specification in Tropos. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering*, Toronto, CA, August 2001.
8. J. Mylopoulos, L. K. Chung, and B. A. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, June 1992.
9. A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.
10. OMG. *OMG Unified Modeling Language Specification*, version 1.3, alpha edition, January 1999.
11. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. In *Proc. of the 5th Int. Conference on Autonomous Agents*, Montreal CA, May 2001. ACM.
12. F. Sannicolo', A. Perini, and F. Giunchiglia. The Tropos modeling language. a User Guide. Technical report, ITC-irst, December 2001.
13. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.
14. E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.