Knowledge Level Software Engineering

Fausto Giunchiglia[1], Anna Perini[2], and Fabrizio Sannicolò[1]

[1] Department of Information and Communication Technology
University of Trento
via Sommarive, 14
I-38050 Trento-Povo, Italy
fausto@dit.unitn.it
sannico@science.unitn.it
[2] ITC-Irst
Via Sommarive, 18
I-38050 Trento-Povo, Italy
perini@irst.itc.it

Abstract. We contend that, at least in the first stages of definition of the early and late requirements, the software development process should be articulated using *knowledge level* concepts. These concepts include *actors*, who can be (social, organizational, human or software) agents, positions or roles, *goals*, and *social dependencies* for defining the obligations of actors to other actors. The goal of this paper is to instantiate this claim by describing how *Tropos*, an agent-oriented software engineering methodology based on knowledge level concepts, can be used in the development of a substantial case study consisting of the meeting scheduler problem.

1 Introduction

In the last few years, many factors, noticeably the higher level of connectivity provided by the network technology and the ever increasing need of more and more sophisticated functionalities, have caused an exponential growth in the complexity of software systems. Examples of application areas where this is the case are e-commerce, e-business and e-services, enterprise resource planning and mobile computing. Software must now be based on open architectures that continuously change and evolve to accommodate new components and meet new requirements. Software must be robust and autonomous, capable of serving users with little or no computer expertise with a minimum of overhead and interference. Software must also operate on different platforms, without recompilations, and with minimal assumptions about its operating environment and its users.

The increased complexity calls for the development of new techniques and tools for designing, developing and managing software systems. We believe that the best way to deal with these problems is to analyze not only the *"what"* and the *"how"* of a software system, but also *"why"* we are using it. This is best done by starting the software development process with the early requirements, where one analyzes the domain within which the system will operate, by studying how the use of the system will modify the environment, and by progressively refining this analysis down to the actual implementation of the single modules. It is our further belief that this kind of analysis can be fruitfully done only by using *knowledge level* notions. In this context, we are using the term "knowledge level" in a technical sense, more precisely in the sense defined by Newell in his Turing Award Lecture [16]. Examples of knowledge level concepts are *actors*, who can be (social, organizational, human or software) agents, positions or roles, *goals*, and *social dependencies* for defining the obligations of actors to other actors. The use of knowledge level notions is necessary in order to analyze how the environment (consisting mainly of human actors) works. Using these notions also in the description of the software modules allows for a uniform and incremental refinement of the notions introduced in the early requirements [3]. From this perspective, agent oriented programming [14, 1, 23] has the advantage of allowing for the use of the same (knowledge level) concepts down to the actual implementation.

In previous papers [20, 19, 10, 11] we have introduced *Tropos* [1], an agent-oriented software development methodology [8, 22] based on the two key ideas hinted above, namely: *(i)* the use of knowledge level 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. Tropos covers five software development phases: *early requirements analysis*, *late requirements analysis*, *architectural design*, *detailed design*, and *implementation*. A core workflow along the whole development process is the *conceptual modeling* activity, performed with the help of a visual modeling language which provides an ontology that includes knowledge level concepts. The syntax of the language is defined through a metamodel specified with a set of UML diagrams [20]. The language provides also a graphical notation for concepts, derived from the $i*$ framework [24] and a set of diagrams for viewing the models properties: *actor diagrams* for describing the network of social dependency relationships among actors, as well as *goal diagrams*, for illustrating goal and plan analysis from the point of view of a specific actor. A subset of the AUML diagrams proposed in [17, 18] are adopted to illustrate detailed design specifications.

The goal of this paper is to instantiate our claim in favour of knowledge level software engineering by describing how Tropos can be used in the development of a substantial case study consisting of the *Meeting Scheduler* problem, as described in [21]. This problem is about the specification of software tools that can support the organization of a meeting, where the preferences of the important attendees are taken into account, as well as the constraints deriving from the other participants' agendas. (See [9, 24] for alternative solutions to this problem.) The paper is structured as follows. The early requirements analysis is described in Section 2, the late requirements analysis in Section 3, and the ar-

[1] From the Greek "tropé", which means "easily changeable", also "easily adaptable". See also [6, 5] for some early work on Tropos.

chitectural design in Section 4. Due to the lack of space, the last two phases, the detailed design and the implementation phases, are quickly summarized in Section 5. Each section starts with a short description of the activities of the phase described and then shows how these activities are instantiated in the solution of the Meeting Scheduler problem. The conclusions are presented in Section 6. A more detailed description of the last two phases can be found in [19].

2 Early Requirements

> *Early Requirements is concerned with the understanding of a problem by studying an existing organizational setting. During this phase, the requirements engineer models the stakeholders as actors and their intentions as goals. Each goal is analyzed from the point of view of its actor resulting in a set of dependencies between pairs of actors.*

The stakeholders of the *Meeting Scheduler* domain, according to the problem statement given in [21], are the following:

- the *Meeting Initiator (MI)*, who wants to organize a meeting (such as a faculty meeting, a project meeting or a program committee meeting) in an effective way. In particular, he/she wants to make sure that the important participants will attend the meeting.
- the *Potential Participant (PP)*, that is, the generic participant, who would like to attend the meeting, possibly according to his/her preferences, or at least avoiding conflicts with other meetings.
- the *Important Participant (IP)*, who is a critical person for the objectives of the meeting. The meeting initiator will take care of checking preferences, both concerning the meeting location and the date, in order to assure his/her presence at the meeting.
- the *Active Participant (AP)*, is a meeting attendee who is required to give a presentation, so he/her possibly needs a overhead-projector or a workstation with or without network connection, and so on.

All these stakeholders can be modeled as roles that can be played by the same person. In particular, the meeting initiator can also play the role of the meeting participant. The complexity of the *Meeting Scheduler* problem is due to the fact that usually several meetings, sharing a set of participants, are going to be organized in parallel.

The initial early requirements model includes the stakeholders (roles) as actors and their intentions as goals. This model is illustrated with the actor diagram depicted in Figure 1, where actors are denoted as circles, goals as ovals and soft goals as cloudy shapes. Soft goals differ from goals because they don't have a formal definition, and are amenable to a different (more qualitative) kind of analysis. Soft goals are most often used to model the system non functional requirements (for instance software qualities, see [7] for a detailed description of soft goals). In particular the actor `MI`, the meeting initiator, has the goal `organize`

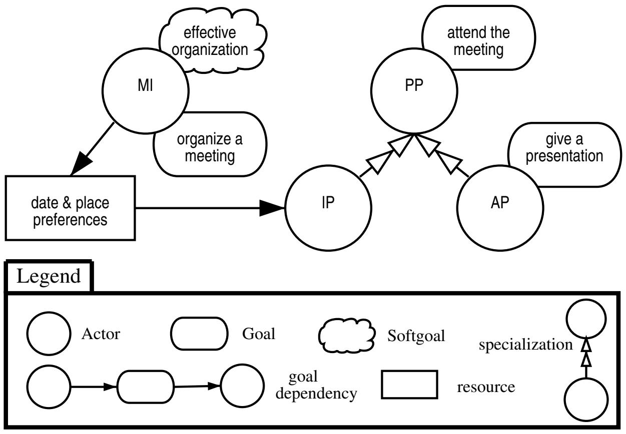

Fig. 1. *Actor diagram* where the social actors of the environment and their goals are modeled.

`a meeting`, possibly in an effective way. This latter objective is modeled as a soft goal, depicted as a cloudy shape labeled `effective organization`. The actor `PP`, models the potential attendee who has the goal `attend the meeting`. The actors `AP` and `IP`, that model the active participant and the important participant respectively, are modeled as a special type of potential participants. [2] That is, both have the goal `attend the meeting`, moreover, the actor `AP` has the goal `give a presentation`. Figure 1 shows also a resource dependency between the actor `MI` (the depender) and the actor `IP` (the dependee), concerning the information on `date & place preferences` of the important participant (the dependum). This dependency makes `MI` vulnerable, in the sense that if `IP` fails to deliver the dependum, then `MI` would be adversely affected in its ability to achieve the objective of arranging the meeting in such a way that the important participant attends it.

The early requirements analysis goes on by extending the actor diagram. This is done by incrementally adding more specific actor dependencies which come out from a goal and plan analysis conducted from the point of view of each actor. Figure 2 depicts a fragment of a goal diagram, obtained by exploding part of the diagram in Figure 1. Here, the perspective of `MI` is modeled. The diagram appears as a balloon within which `MI`'s goals are analyzed and his/her dependencies with the other actors are established. The goal `organize a meeting` is AND-decomposed into the subgoals `define meeting objectives`, `plan the meeting` and `identify participants`. Means to the achievement of the goal `plan the meeting` are the following two goals: `collect requirements`

[2] At the moment, it is still an open problem whether this kind of construct will appear in the final version of the methodology. The question is whether this construct is really a knowledge level construct or, rather, a software level construct that we are "forcing" in the early requirements analysis.

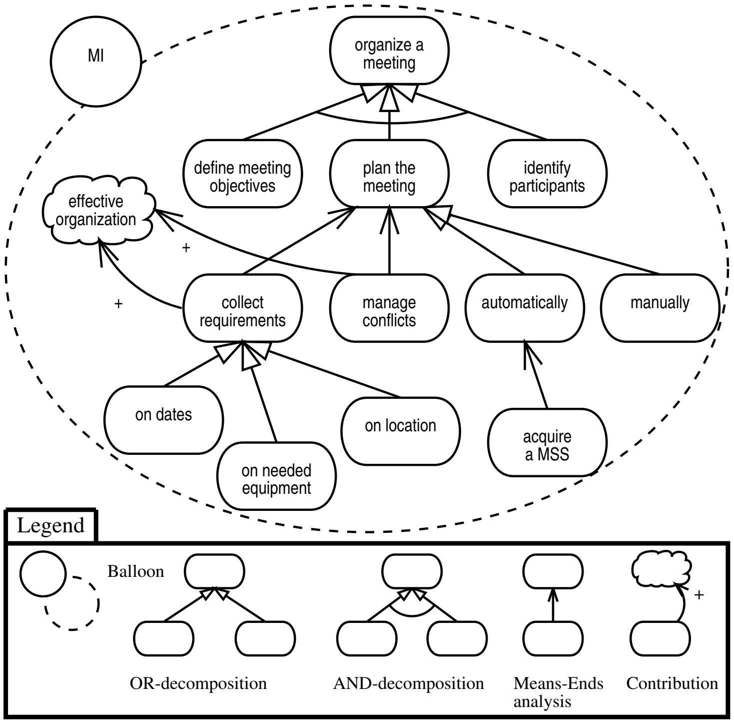

Fig. 2. Goal diagram for the actor `Meeting Initiator`, focusing on its goal `organize a meeting`.

and `manage conflicts`. The goal `collect requirements` refers to the need of asking the participants for the following information based on their personal agenda: dates on which they cannot attend the meeting (the "exclusion set"), and dates on which they prefer the meeting to take place (the "preference set"). Moreover, the active participants should specify the need of specific equipment for their presentation, while important participants could indicate their preference on the meeting location. So, the goal `collect requirements` has been further OR-decomposed into the goals `on dates`, `on needed equipment` and `on location`. According to [21], conflicts can be solved in several ways, such as extending the initial date range provided by the meeting initiator, asking participants to remove some dates from their exclusion set, asking important participant to add some dates to their preference set or withdrawing some participants from the meeting.

The goal `plan the meeting` could be achieved with the help of a software tool (see subgoal `automatically`) or manually (see subgoal `manually`). The goal `acquire a MSS`, that is the goal of acquiring a Meeting Scheduler System (MSS), is a means for achieving the goal of using a software tool. Both the goals `manage`

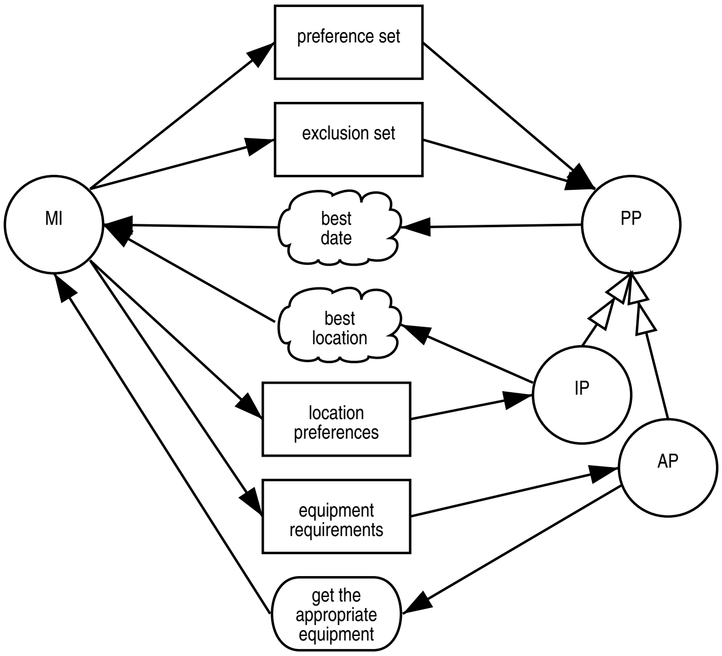

Fig. 3. Actor diagram showing the main dependencies between the social actors.

`conflicts` and `collect requirements` contribute positively to the fulfillment of the soft goal `effective organization`.

Figure 3 depicts the actor diagram resulting from the completion of the goal analysis of each social actor. `MI` depends on `PP`, the potential participant, for the information `preference set` and `exclusion set`, modeled as resources. Vice-versa, `PP` depends on `MI` for satisfying the soft goal `best date`. Moreover, `MI` depends on `IP` for getting preferences on the meeting location (see the resource dependency `location preferences`), and, vice-versa, `IP` depends on `MI` for the fulfillment of the soft goal `best location`. Finally, `MI` depends on `AP` for knowing the requirements on the specific equipment needed for the presentation, and `AP` depends on `MI` for the goal `get the appropriate equipment`.

3 Late Requirements

The Late Requirements phase analyses the system-to-be which is introduced as another actor into the model. The system actor is related to the social actors in terms of actor dependencies; its goals are analyzed and will eventually lead to revise and add new dependencies with a subset of the social actors (the users). When the scenario is sufficiently detailed, this provides a "use-cases" view.

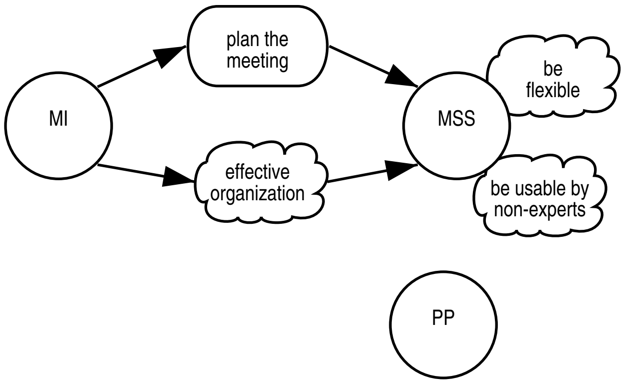

Fig. 4. Actor diagram where the system-to-be actor has been introduced.

The system-to-be, that is the *Meeting Scheduler System*, is represented by the actor `MSS`. Figure 4 illustrates the late requirements actor diagram. `MI` depends on the actor `MSS` for the goal `plan the meeting`, one of the `MI`'s sub-goals discovered during the goal analysis depicted in Figure 2. The soft goal `effective organization` has been also delegated by `MI` to `MSS`. The dependencies between the social actors illustrated in Figure 2 need to be revised accordingly. The actor `MSS` has also two soft goals: `be flexible` and `be usable by non-experts`, which take into account some of the non-functional requirements described in [21].

The balloon in Figure 5 shows how the `MI`'s dependums can be further analyzed from the point of view of the *Meeting Scheduler System*. The goal `plan the meeting` is decomposed (AND-decomposition) into the sub-goals `identify location`, `identify possible dates`, `communicate date & location` to the `IP`. The goal `plan the meeting` is a means for obtaining the goal `replan dynamically`, which is introduced as a goal of the actor `MSS`, according to the requirements of the *Meeting Scheduler System*. The goal `replan dynamically` contributes to the soft goal `be flexible`. The analysis proceeds by further decomposing goals into sub-goals and by identifying plans that can be considered as means for achieving these latter goals. So, for instance the goal `identify location` is AND-decomposed into the sub-goals `get equipment requirements`, `know location preferences`, `check location availability`, `solve possible conflicts`. Analogously, the goal `identify possible dates` is AND-decomposed into the sub-goals `manage conflicts`, `define exclusion sets` and `define preference sets`. The plan `get conflict management policies` from the meeting initiator has been introduced as a means for achieving the leaf goal. Analogously, the plans `get new exclusion set`, `get new preference set`, `get new location preferences`, are all means for achieving the goal `replan dynamically`.

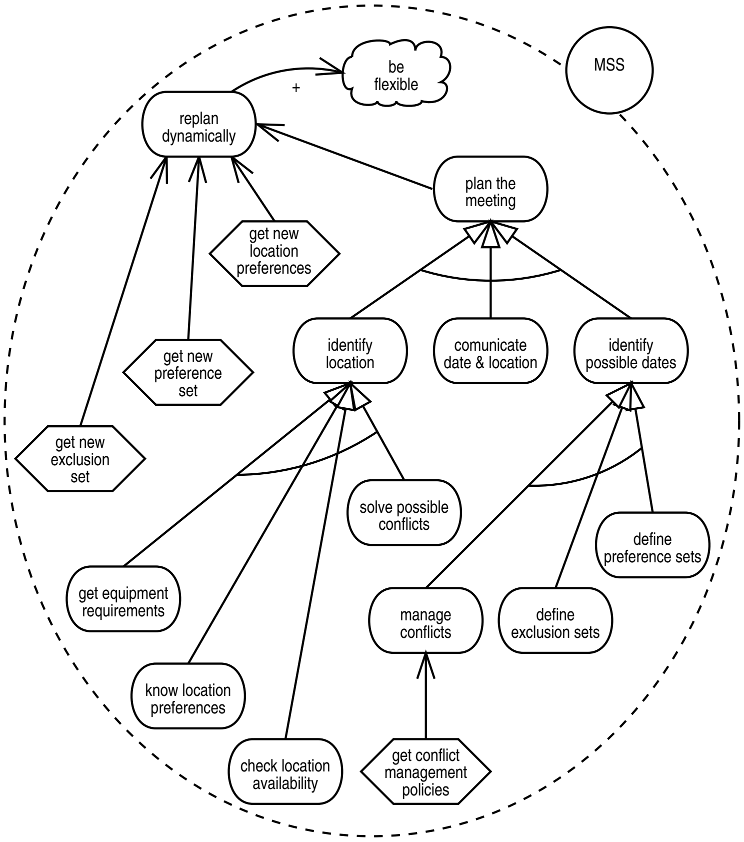

Fig. 5. Goal diagram of the system-to-be actor.

A set of dependencies between the system-to-be actor and the social actors can be derived from this analysis, as shown in the actor diagram depicted in Figure 6. So, the actor `MI` delegates to the system actor `MSS` the goal `plan the meeting`, while `MSS` depends on the actor `MI` for having a set of `conflict resolution policies` and general information about the meeting, such as the meeting objectives and information necessary to identify potential, important and active participants. Moreover, the plans identified in the goal analysis of the `MSS` actor motivates a set of resource dependencies among the system actors and the actors modeling the different roles of the meeting participants, that is `date preferences`, `location preferences`, `exclusion set`, `equipment requirement`. Vice-versa, the following resource dependencies link these actors and the `MSS` actor: `new date preference`, `new location preference`, `new exclusion set` and `new equipment requirement`. Finally, `PP` depends on `MSS` for being promptly advised about the `final date & location` of the meeting. The resulting actor diagram sketches a “use-case”

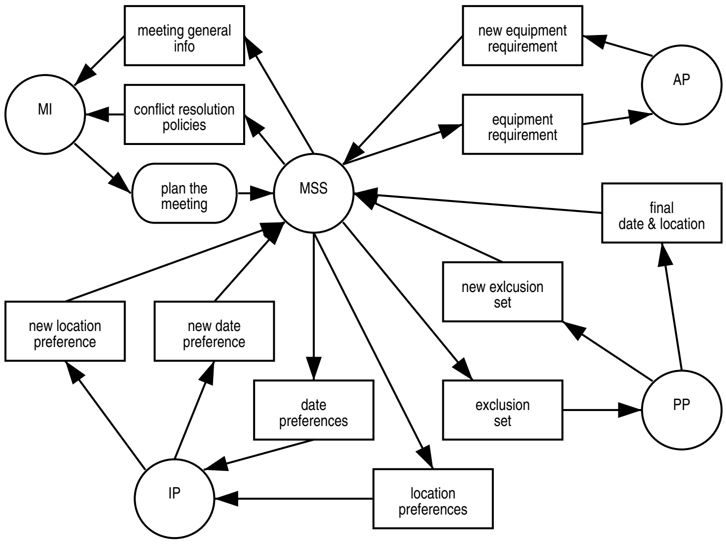

Fig. 6. Actor diagram resulting from the late requirements analysis.

diagram [13]. Notice how here use-cases are obtained as a refinement of the Early and Late requirements Analysis. Notice also how this is not the case for UML [2] where use-case diagrams are an add-on which hardly integrates with the other diagrams.

4 Architectural Design

Architectural design defines the system's global architecture in terms of subsystems, interconnected through data and control flows. Subsystems are represented as actors and data/control interconnections are represented as (system) actor dependencies. This phase consists of three steps:

1. *refining the system actor diagram,*
2. *identifying capabilities and*
3. *assigning them to agents.*

Step 1. The actor diagram is refined adding new sub-systems, according to the following sub-steps:

a inclusion of new actors due to the delegation of sub-goals, upon goal analysis of the system's goals;

b inclusion of new actors according to the choice of a specific architectural style agent (design patterns [15]);

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

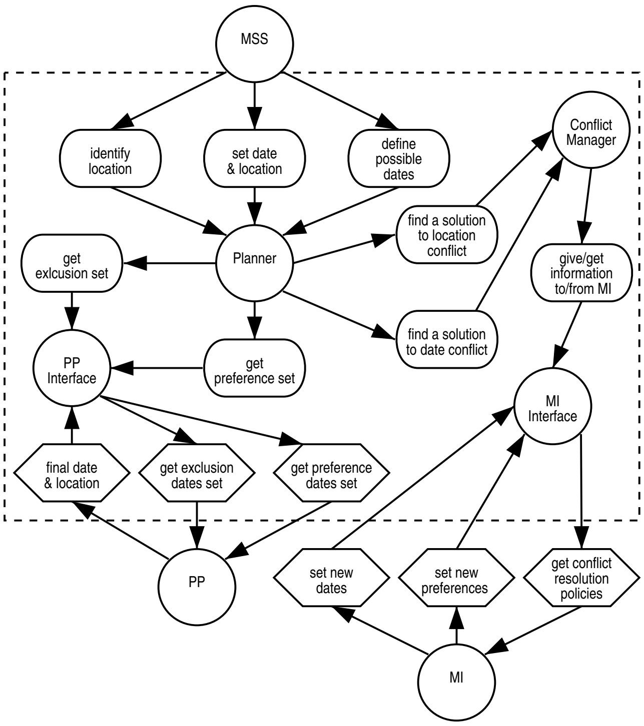

Fig. 7. A Portion of the architectural design model for the system-to-be. Extended actor diagram.

Here, we focus on an example of step 1.a. A portion of the system architecture model is illustrated in the actor diagram depicted in Figure 7. Four sub-system actors have been introduced. The first is the `Planner` to which `MSS` delegates the following goals: `identify location`, `define possible dates`, `set date & location`. The `Planner` rests on the actor `PP Interface` for the achievement of the goals `get exclusion set`, `get preference set`, and on the actor `Conflict Manager` for the achievement of the goals `find a solution to location conflict` and `find a solution to date conflict`. Moreover, the actor `Conflict Manager` rests on the actor `MI Interface` to satisfy its goal of giving and getting information to/from the meeting initiator. Figure 7 includes also some dependencies with the social actors interacting with the system. In particular, the actor `PP Interface` depends on the social actor `PP`, to have the plans of getting exclusion and preference dates set, while `PP` depends on the actor `PP Interface` to receive information on the meeting (such as final date and location). Analogously, the actor `MI Interface` depends on the social actor `MI` to execute the plans of getting conflict resolution policies, and, vice-versa, `MI` depends on the actor `MI Interface` to be be able to set new preferences and new dates for the meeting to be organized.

Actor Name	N	Capability
Planner	1	identify location
	2	define possible dates
	3	set date & location
	4	get exclusion set
	5	get preference set
	6	find a location conflict solution
	7	find a date conflict solution
PP Interface	8	provide exclusion set
	9	provide preference set
	10	interact with PP to get exclusion set
	11	interact with PP to get preference set
Conflict Manager	12	resolve a conflict
	13	give/get information to/from MI
	14	provide solutions to location conflicts
	15	provide solutions to date conflicts
MI Interface	16	get & provide info to Conflict Manager
	17	interact with MI to get resolution policies
	18	interact with MI to get new dates
	19	interact with MI to get new preferences

Table 1. Actors capabilities.

The system architecture model can be further enriched with other system actors according to design patterns [12] that provide solutions to heterogeneous agents communication and to non-functional requirements, as also described in [10].

> *Step 2. The actor capabilities are identified from the analysis of the dependencies going-in and -out from the actor and from the goals and plans that the actor will carry on in order to fulfill functional and non-functional requirements.*

Focusing on the system actor `Planner` depicted in Figure 7, and in particular on its ongoing and outgoing dependencies we can identify the following capabilities: `identify location`, `define possible dates`, `set date & location`, `get exclusion set`, `get preference set`, `find a location conflict solution`, `find a date conflict solution`. Table 1 lists the capabilities associated to the actors depicted in the diagram of Figure 7.

> *Step 3. Agent types are defined. One or more different capabilities are assigned to each agent.*

Agent	Capabilities
Planner	1, 2, 3, 4, 5, 7
User Interface	6, 8, 9, 10, 11, 16, 17, 18, 19
Conflict Manager	12, 13, 14, 15

Table 2. Agent types and their capabilities. An example.

In general, the agent assignment is not unique and depends on the designer experience. Table 2 shows an example of agent assignment on a subset of the system-to-be actors. This table shows that there is more than one mapping from actors and agent types. Other associations could be easily find out.

5 The last two phases: Detailed Design and Implementation

Detailed design aims at specifying the agent microlevel, defining capabilities, and plans using the AUML activity diagram, and communication and coordination protocols using the AUML sequence diagrams. A mapping between the Tropos concepts and the constructs of the implementation language and platform is provided.

Figure 8 depicts, as an example, the capability diagram of the `manage location conflict`, one of the capabilities of the agent `Conflict Manager`. The capability is triggered by the external event corresponding to the request of the agent `Planner` to the `Conflict Manager` of managing a location conflict, given a list of preferred locations and a list of available time periods for each location. The agent `Conflict Manager` chooses one of the possible plan corresponding to different conflict management policies. For instance, `policy 1` corresponds to the plan which, for a given date, checks the availability of the meeting location, according to the stated preference order. If there is no solution the `Conflict Manager` tries a policy that has not yet been used or asks the user for a solution, via the `User Interface actor`. The plan `policy 2` checks, for a given location, the dates at which the location is not allocated, taking into account the preferred date first. If there is no solution the `Conflict Manager` tries a policy that has not yet been used or asks the user for a solution, via the `User Interface` agent. The third policy corresponds to the plan where the user is asked to: (i) sort the preferred dates, (ii) sort the preferred locations, (iii)

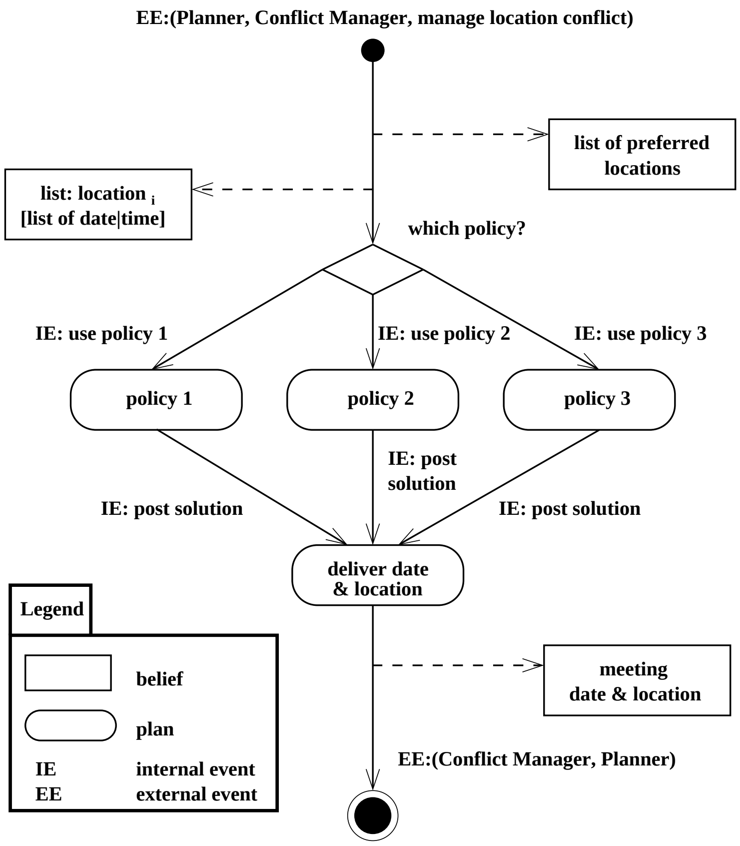

Fig. 8. *Capability diagram* of `manage location conflict`, one of the capabilities of the actor `Conflict Manager`.

choose which policy, among `policy 1` and `policy 2` should be tried first[3]. The solution is noticed back to the agent `Planner`.

> *Finally, the implementation activity follows step by step, in a natural way, the detailed design specification which is transformed into a skeleton for the implementation.*

This is done through a mapping from the Tropos concepts to the constructs of the programming language provided by the implementation platform that has been chosen before starting the detailed design phase. An example, based on the use of the BDI agent programming platform JACK [4], can be found in [19].

[3] *Plan diagrams*, namely AUML activity diagrams, for the given plans can be straightforwardly derived. They are not included here due to lack of space. For the same reason we can not show here the specification of agent interactions via AUML sequence diagram, see [19] for an example.

6 Conclusions

In this paper we have applied the Tropos methodology to the case study of the Meeting Scheduler problem. Tropos is a new software development methodology, mainly designed, but not only, for agent based software systems, which allows us to model software systems at the knowledge level. The main goal of this paper has been to show, via an example, how, by working at the knowledge level, we can cope with the increased complexity that many new applications require.

7 Acknowledgments

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

References

1. FIPA. Foundation for Intelligent Physical Agents. http://www.fipa.org.
2. G. Booch, J. Rambaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.
3. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: a transformation based approach. In Wooldridge et al. [22].
4. 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.
5. J. Castro, M. Kolp, and J. Mylopoulos. Developing agent-oriented information systems for the enterprise. In *Proceedings Third International Conference on Enterprise Information Systems*, Stafford UK, July 2000.
6. J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proc. 13th Int. Conf. on Advanced Information Systems Engineering CAiSE 01*, Stafford UK, June 2001.
7. L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
8. P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*. Springer-Verlag, March 2001.
9. R. Darimont, A. van Lamsweerde, and P. Massonet. Goal-Directed elaboration of requirements for a meeting scheduler: problems and lesson learnt. In *Proceedings RE'95 - 2nd IEEE Symposium on Requirements Engineering*, pages 194–203, York, March 1995.
10. P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. In S. Sen J.P. Müller, E. Andre and C. Frassen, editors, *Proceedings of the Thirteenth International Conference on Software Engineering - Knowledge Engineering (SEKE01)*, Buenos Aires - ARGENTINA, June 13 - 15 2001.
11. F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. Technical Report No. 0111-20, ITC - IRST, Nov 2001. Submitted to AAMAS '02.

12. Sandra Hayden, Chirstina Carrick, and Qiang Yang. Architectural design patterns for multiagent coordination. In *Proc. of the International Conference on Agent Systems '99*, Seattle, WA, May 1999.
13. Ivar Jacobson, Mangus Christerson, Patrik Jonsson, and Gunmar Övergaard. *Object-Oriented Software Engineering: a Use-Case Driven Approach.* Addison Wesley, Readings, MA, 1992.
14. N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2), 2000.
15. M. Kolp, P. Giorgini, and J. Mylopoulos. An goal-based organizational perspective on multi-agents architectures. In *Proc. of the 8th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Seattle, WA, August 2001.
16. A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.
17. J. Odell and C. Bock. Suggested UML extensions for agents. Technical report, OMG, December 1999. Submitted to the OMG's Analysis and Design Task Force in response to the Request for Information entitled "UML2.0 RFI".
18. J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proc. of the Agent-Oriented Information Systems workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, Austin, TX, 2000.
19. 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.
20. F. Sannicolo', A. Perini, and F. Giunchiglia. The Tropos modeling language. a User Guide. Technical report, ITC-irst, December 2001.
21. A. v. Lamsweerde, R. Darimont, and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), October 1992.
22. M. Wooldridge, P. Ciancarini, and organizers G. Weiss, editors. *Proc. of the 2nd Int. Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, CA, May 2001.
23. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.
24. E. Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Department of Computer Science, 1995.