Corso di Laurea Specialistica in Informatica

TESI DI LAUREA

Knowledge Level Engineering of BDI Agents
Knowledge level engineering di agenti BDI

Relatore: Dr. Paolo Giorgini

Laureando: Mirko Morandini

Correlatori: Dr. Loris Penserini
Dr. Angelo Susi

Anno Accademico 2005/2006
# Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduzione</td>
<td>7</td>
</tr>
<tr>
<td>Obiettivi della tesi</td>
<td>8</td>
</tr>
<tr>
<td>Approccio</td>
<td>8</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Thesis Objectives</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Approach</td>
<td>13</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>14</td>
</tr>
<tr>
<td>2 Agents - State of the Art</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Software Agents</td>
<td>15</td>
</tr>
<tr>
<td>Differences to OOP</td>
<td>16</td>
</tr>
<tr>
<td>“Intelligent” or “rational” Agents</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1 The BDI model</td>
<td>17</td>
</tr>
<tr>
<td>2.1.2 Multi-Agent Systems</td>
<td>18</td>
</tr>
<tr>
<td>2.2 AOSE Methodologies</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1 MaSE</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1.1 The development phases</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2 Gaia</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2.1 The development phases</td>
<td>24</td>
</tr>
<tr>
<td>2.2.3 Passi</td>
<td>25</td>
</tr>
<tr>
<td>2.2.3.1 Models and development phases</td>
<td>26</td>
</tr>
<tr>
<td>2.2.4 MAS-CommonKADS</td>
<td>27</td>
</tr>
<tr>
<td>2.2.4.1 Models and development phases</td>
<td>27</td>
</tr>
<tr>
<td>Conceptualization phase</td>
<td>27</td>
</tr>
<tr>
<td>Analysis phase</td>
<td>28</td>
</tr>
<tr>
<td>Design phase</td>
<td>29</td>
</tr>
<tr>
<td>2.2.5 Prometheus</td>
<td>30</td>
</tr>
<tr>
<td>2.2.5.1 The development phases</td>
<td>30</td>
</tr>
<tr>
<td>System Specification phase</td>
<td>30</td>
</tr>
<tr>
<td>Architectural Design phase</td>
<td>31</td>
</tr>
<tr>
<td>Detailed Design phase</td>
<td>32</td>
</tr>
<tr>
<td>2.2.5.2 Tool support and final considerations</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Agent Platforms</td>
<td>34</td>
</tr>
<tr>
<td>2.3.1 Jade</td>
<td>35</td>
</tr>
<tr>
<td>Agents in Jade</td>
<td>35</td>
</tr>
<tr>
<td>Platform architecture</td>
<td>36</td>
</tr>
<tr>
<td>Tools</td>
<td>36</td>
</tr>
<tr>
<td>2.3.2 Jadex</td>
<td>37</td>
</tr>
<tr>
<td>Beliefs</td>
<td>38</td>
</tr>
<tr>
<td>Plans</td>
<td>38</td>
</tr>
<tr>
<td>Goals</td>
<td>38</td>
</tr>
<tr>
<td>Tool support</td>
<td>40</td>
</tr>
</tbody>
</table>
2.4 TROPOS .................................................................................................................. 40
  2.4.1 Key concepts ........................................................................................................ 41
  2.4.2 Graphical notation ............................................................................................... 42
  2.4.3 The development phases ...................................................................................... 44
    Early Requirements analysis ...................................................................................... 44
    Late Requirements Analysis ....................................................................................... 45
    Architectural Design .................................................................................................... 45
  2.4.4 The new Capability definition .............................................................................. 45
  2.4.5 Formal Tropos ..................................................................................................... 47
  2.4.6 TAOM4E ............................................................................................................. 48

3 Methodological Approach ........................................................................................... 49
  Agents and Roles ........................................................................................................... 50
  3.1 The knowledge level in TROPOS .......................................................................... 50
  3.2 The knowledge level at run-time ............................................................................ 51
    3.2.1 What we are trying to achieve – and what not ..................................................... 52
  3.3 Preparing the TROPOS concepts .......................................................................... 53
    3.3.1 The diminished set of TROPOS relationships ..................................................... 54
    Combining relationships ............................................................................................. 55
    3.3.2 Semantics, approaching an implementation ....................................................... 55
      AND-Decomposition ................................................................................................. 55
      OR-Decomposition ................................................................................................... 56
      Means-End ............................................................................................................... 56
      Use and Produce ...................................................................................................... 57
      Contribution ............................................................................................................. 57
      Delegation ................................................................................................................ 58
      'why'-Dependency ................................................................................................... 58

4 Mapping and Implementation ................................................................................... 61
  4.1 Specifications and Requirements .......................................................................... 61
    Requirements on the mapping .................................................................................... 62
    Requirements on the generated code ......................................................................... 62
    Requirements on the code generation tool ................................................................. 63
  4.2 The mapping ........................................................................................................... 63
    4.2.1 Characteristics of Jadex, focusing on the mapping .............................................. 63
    4.2.2 Entities mapping ............................................................................................... 64
      Plan ........................................................................................................................... 64
      Goal .......................................................................................................................... 65
      Goal types ............................................................................................................... 65
      Resource .................................................................................................................. 65
      Softgoal .................................................................................................................... 66
    4.2.3 Relationships mapping ...................................................................................... 66
      Means-end relation ................................................................................................... 66
      OR-decomposition ................................................................................................... 67
      AND-decomposition ............................................................................................... 69
      Contribution ............................................................................................................. 69
      Delegation ............................................................................................................... 69
Figures

Figure 2.1.: The development steps of the MaSE methodology (from [Wood and DeLoach, 2000]).
Figure 2.2.: Sample MaSE diagrams for the Architectural Design phase (from [Wood and DeLoach, 2000]).
Figure 2.3.: Sample MaSE diagrams for the Detailed Design phase (from [Wood and DeLoach, 2000]).
Figure 2.4.: Relationships between Gaia models (from [Wooldridge et al., 2000]).
Figure 2.5.: The models and phases of the PASSI methodology (from [Cossentino, 2005]).
Figure 2.6.: Organizational model class diagram notation (as in [Iglesias and Garijo, 2005]).
Figure 2.7.: Overview on the Prometheus development phases and artefacts (from [Sudeikat, 2004]).
Figure 2.8.: Prometheus system overview diagram (from [Sudeikat, 2004]).
Figure 2.9.: Prometheus agent overview diagram (from [Sudeikat, 2004]).
Figure 2.10.: Definition of a Jadex agent (from [Pokahr and Braubach, 2005]).
Figure 2.11.: Jadex Sniffer Agent for graphical message tracking.
Figure 2.12.: Reference architecture of a FIPA Agent Platform.
Figure 2.13.: Jadex goal life-cycle (from [Pokahr and Braubach, 2005]).
Figure 2.14.: (A): Graphical representation of the meta-reasoning process. (B): Concise representation used in this thesis.
Figure 2.15.: Part of the TROPOS metamodel, focusing on the goal concept (from [Sannicolò et al., 2001]).
Figure 2.16.: Graphical representation of the TROPOS concepts.
Figure 2.17.: Decompositions and contribution.
Figure 2.18.: TROPOS actor dependencies.
Figure 2.19.: Example diagrams for the first four phases of the TROPOS methodology (from [Bertolini et al, 2005]).
Figure 2.20.: Capability table for a sample agent, generated by Taom4E.
Figure 2.21.: Capability: (A) complete view in Tropos notation, (B) activity diagram for control-flow structure of the ability plan, and (C) agent-interaction for one activity of the ability (from [Penserini et al., 2006a]).
Figure 2.22.: Taom4e, embedded into an Eclipse window, with an example Tropos diagram.
Figure 3.1.: Example TROPOS diagram with knowledge level and capabilities with abilities (plan level) highlighted.
Figure 3.2.: Modelling relationship combinations in the diminished Tropos model.
Figure 3.3.: Allowed AND-decompositions.
Figure 3.4.: Allowed OR-decompositions.
Figure 3.5.: The means-end relationship and the resource use and produce relationships.
Figure 3.6.: Allowed contribution relationships.
Figure 3.7.: Goal delegation from a stakeholder to an actor in the system.
Figure 4.1.: Possibilities for goal dispatching and plan triggering in Jadex....................... 64
Figure 4.2.: The mapping of the means-end relation from TROPOS to Jadex.................. 66
Figure 4.3.: The mapping of the OR-decomposition relation from TROPOS to Jadex. ... 68
Figure 4.4.: Considered equivalence of contributions to different levels in the goal hierarchy.................................................................................................................. 68
Figure 4.5.: The mapping of the AND-decomposition relation from TROPOS to Jadex. ................................................................................................................................. 69
Figure 4.6.: Example mapping of a 'why'-dependency relation from TROPOS (drawn with Taom4E) to Jadex......................................................................................... 71
Figure 4.7.: An example: TROPOS goal diagram of a simple search agent..................... 73
Figure 4.8.: Graphical representation of the Jadex architecture for the simple search agent depicted in Figure 4.7. ................................................................. 74
Figure 4.9.: Class diagram of the plan classes used in the mapping. Aplan and AmetaPlan extend jadex.runtime.Plan.......................................................... 82
Figure 4.10.: Class diagram displaying the classes used for belief base objects mapped from Tropos components............................................................................... 83
Figure 5.1.: Requirements analysis (simplified) for the SearchSystem............................ 87
Figure 5.2.: Architectural Design diagram with focus on goal decomposition of the SearchActor, drawn with Taom4e................................................................. 89
Figure 5.3.: Goal decomposition for “find word description” and graphical run-time representation of the agent’s internal structure, generated by the Jadex Introspector tool.......................................................................................................................... 92
Figure 5.4.: Inter-agent communication for the achievement of the goal “find copied text”. Left side: Simple test agent for user requests. Right side: Jade Sniffer Agent................................................................................................................................. 93
Figure 5.5.: Expansion of agent goal trees during the achievement process for the goal “find copied text”.......................................................................................... 94
Figure 5.6.: Completion of the goal achievement process started in Figure 5.5 (screenshots from the Jadex Introspector)................................................................. 94
Introduzione

Con la diffusione di Internet ed una potenza di calcolo disponibile sempre crescente, al giorno d'oggi sempre più applicazioni sofisticate sono richieste per web services, reti peer to peer, sistemi ERP ed in aree emergenti come l'ubiquitous computing. Queste applicazioni devono operare in ambienti aperti, eterogenei ed in continuo cambiamento. Per soddisfare i bisogni di una vasta gamma di utenti, sistemi software devono essere robusti, devono poter operare sotto condizioni incerte ed essere abili a svilupparsi per adattarsi alle esigenze degli utenti, assicurando qualità di servizio invariate.

Il paradigma di programmazione ad agenti (Agent Oriented Programming, AOP) è perfettamente adatto per affrontare la complessità di questo tipo di applicazioni. Agenti sono componenti software autonome che sanno interagire e cooperare. Con i loro concetti di goal, stati mentali, conoscenza ed abilità sociale si definiscono un livello d'astrazione più alto di oggetti tradizionali con classi, metodi e stati ben definiti. Così il concetto di agente aiuta ad implementare e amministrare sistemi flessibili, distribuiti ed interdipendenti.

Per catturare i concetti dell'AOP nel processo di sviluppo di un sistema, è necessaria una metodologia di ingegneria del software orientata ad agenti (AOSE methodology). Diverse metodologie sono state sviluppate per arrivare dalle intenzioni degli stakeholder alla progettazione di architettura ed implementazione di un sistema ad agenti, tra le più conosciute ci sono Gaia, MaSE, MAS-CommonKADS, Passi, Prometheus e TROPOS, presentati nel capitolo 2. Ognuna di queste metodologie addotta diversi concetti di alto livello, per affrontare requisiti complessi e per sostenere le diverse fasi di design di un software. Diversamente da metodologie tradizionali orientate ad agenti, le applicazioni sono concepite non come componenti software, ma molto più come organizzazioni sociali, descritte come sistemi multi-agente – società di agenti collaborativi senza controllo centralizzato – definendo responsabilità, dipendenze, interazioni e funzionalità, le quali un agente deve fornire.
**Obiettivi della tesi**

La metodologia TROPOS copre le fasi di sviluppo di un software dall’analisi dei requisiti di base al design dettagliato, usando una rappresentazione esplicita per agenti e nozioni mentalistiche correlate come goal, piano, e dipendenza. Una parte importante dell’analisi viene trasferita da un livello orientato ad oggetti (progettando classi, metodi ed attributi) ad un livello di conoscenza (knowledge level), analizzando le richieste degli stakeholder, delegando e decomponendo i goal dei diversi attori, modellando la loro conoscenza e definendo dipendenze tra di loro. Le funzionalità che il sistema deve esibire, le capability, vengono derivate da questa analisi. Tutte le informazioni raccolte fino a questo punto devono poi condurre ad un’implementazione del sistema richiesto.

Per molte metodologie sono stati definiti dei mapping dal design all'implementazione, ma molti di questi riguardano solo la parte operativa, le funzionalità di un sistema, non tenendo conto delle informazioni a livello di conoscenza attinenti, modellate nelle prime fasi del processo di sviluppo. Noi sosteniamo che per un agente è importante essere conscio delle informazioni recuperate dall’analisi del knowledge level effettuata durante il processo di sviluppo, per dargli l’abilità di riflettere su quali funzionalità eseguire e su come e quando farlo, per soddisfare le esigenze degli stakeholder. Per questo, le informazioni devono essere propagate attraverso le fasi successive di design, fino all'implementazione.

L'obiettivo della tesi è di superare questo divario tra il design e l'implementazione. Viene sviluppando un framework per agenti razionali che si comportano secondo le intenzioni degli stakeholder, definendo una mappatura ad una piattaforma ad agenti ed un tool per una trasformazione automatica. L’approccio proposto garantisce di produrre agenti capaci di reagire ad eventualità in modo conforme ai requisiti e, vice versa, dà la possibilità di far salire decisioni fatte all’implementazione e durante l’esecuzione, fino alle fasi di definizione dei requisiti.

**Approccio**

Questa tesi propone un approccio basato sulla metodologia TROPOS, estesa con la tecnica del “capability modelling” [Penserini et al., 2006a][Penserini et al., 2006b] per separare il knowledge level dalle concrete funzionalità dell’agente. Le funzionalità possono essere modellati decomponendo piani in diagrammi TROPOS seguendo metodi object-ori-
entato usando activity diagram e sequence diagram di UML 2.0 e generando codice usando un tool specifico.

Per raggiungere un goal, un agente deve essere consapevole dell'informazione a livello di conoscenza estratta dai modelli TROPOS, per essere capace di scegliere i subgoal da raggiungere e le migliori funzionalità da eseguire, secondo lo stato attuale dell'ambiente, delle condizioni di contesto e della contribuzione ad altri goal, monitorando errori ed amministrando loro recupero.

Queste esigenze portano ad un'implementazione come “agente razionale”, usando un'architettura BDI (Belief-Desire-Intention, [Rao and Georgeff, 1991]). In particolare, nella tesi si adotta Jadex [Pokahr et al., 2005], una piattaforma ad agenti conforme con le specifiche FIPA e basata su architettura BDI.

L’approccio proposto definisce un mapping diretto ed automatico di goal, dipendenze e decomposizioni di goal dalla fase di Detailed Design di TROPOS al codice, per correlare il knowledge level con le istanze di agenti software. Ad un set di strutture TROPOS a Detailed Design ridotto è stata associata una semantica ben definita, per poter caratterizzare un framework di agenti, con focus su concetti BDI. Il framework definito mira più ad essere flessibile ed adattabile che a includere tecniche specifiche di intelligenza artificiale.

Basando su questo mapping, è stato implementato un prototipo per la generazione automatica di codice ad agenti da diagrammi TROPOS, estendendo il design-tool TAOM4E. Questo prototipo, chiamato t2x, produce strutture eseguibili di agenti basati su Jadex, riducendo di molto il tempo di implementazione dei concetti modellati.
1 Introduction

Nowadays, with the widespread use of Internet and ever growing computational power, more and more sophisticated software applications are demanded in areas such as web services, peer-to-peer networks, enterprise resource planning, and ubiquitous computing. These applications have to operate within open, evolving and heterogeneous environments. To satisfy the needs and intentions of a wide range of users, systems require to be robust and highly customizable, to operate under uncertain conditions and to react flexibly to failures. Furthermore, they should be able to adapt autonomously to modified circumstances and to evolve over time to cope with changing requirements, assuring users’ quality-of-service demands.

The Agent Oriented Programming paradigm (AOP) is best suited to deal with the increasing complexity of such applications. Software Agents are autonomous and goal-driven software components that can interact and cooperate. Agents define a higher level of abstractions from low-level technologies than objects, promoting modularity and decomposition to autonomous units [Jennings, 2001]. In AOP, we talk of mental states and beliefs instead of machine states, of plans and actions instead of procedures and methods, of communication and social ability instead of I/O functionalities [Bresciani et al, 2004]. These concepts help to implement and manage robust, flexible, distributed, and interwoven software systems like the ones mentioned above.

All these concepts need to be captured by a software engineering methodology in order to properly move towards the underlying software agent implementation; promising results are coming from the area of Agent Oriented Software Engineering (AOSE) methodologies.

AOSE methodologies such as Gaia [Zambonelli et al., 2003] [Wooldridge et al., 2000], MaSE [DeLoach and Sparkman, 2001], MAS-CommonKADS [Iglesias et al., 1998], PASSI [Cossentino and Potts, 2002], Prometheus [Padgham and Winikoff, 2002], and TROPOS [Castro et al., 2002] [Bresciani et al., 2004] support an analysis process in order to figure out stakeholders’ intentions (requirements) and to show how such requirements affect the architectural design and implementation phases of agent oriented systems. The different methodologies adopt different high level
design concepts in order to abstract from complex system requirements and to support
different design phases. The most promising of them are treated in Section 2.2. In con-
trast to traditional object-oriented software engineering methodologies (e.g. RUP,
OMT), applications are conceived much more like social organizations than as software
components. Software systems are described as multi-agent systems (MAS) – societies of
collaborative agents without centralized control – defining responsibilities, dependen-
cies, interactions, and the system functionalities the single agents have to provide.

1.1 Thesis Objectives

The TROPOS methodology covers the software development phases from the early re-
quirements analysis to the detailed design, using an explicit representation of Agents
and the related mentalistic concepts of Goal, Plan, and Dependency. An important part of
the analysis is moved from the object-oriented software level (designing classes, attrib-
utes, and methods) at a knowledge level, analysing stakeholders' desires, delegating and
decomposing actors' goals, modelling each actor's knowledge of the environment, and
defining dependencies among actors. The system functionalities, which may correspond
to the agent capabilities, are then derived from this analysis. At this point, all the collected
information has to conduct to an implementation of the software.

Although for several AOSE methodologies, mappings from detailed design to the
implementation were proposed, most of them are only concerned with the operative
part of system functionalities, without accounting about related knowledge level inform-
ation figured out at the early phases of design. We claim that software agents at run-
time have to be conscious about the knowledge level analysis performed during the
design process, to give them the ability to reason on what functionalities to perform, and
when and why to do it, to satisfy stakeholders' needs. Therefore, knowledge level inform-
ation has to be propagated towards the later phases of design till the implementation.

The objective of this thesis is to overcome this semantic gap between
architectural/detailed design and implementation phases, accounting for agent know-
ledge about the domain and for how stakeholder intentions may influence agent
behaviours. A framework for autonomous and social reasoning agents that behave accord-
ing to the stakeholders’ intentions is defined, developing mappings to concepts of an
agent-oriented platform and a transformation tool. The proposed approach guarantees
to produce agents able to react to eventualities in a way that meets the designed require-
1.1 Thesis Objectives

ments and, vice versa, gives the possibility to backtrack agent decisions, made at implementation- and run-time, up to the requirements engineering phases.

1.2 Approach

This thesis proposes an approach based on the TROPOS methodology, extended with the agent capability modelling technique as detailed in [Penserini et al., 2006a][Penserini et al., 2006b], to separate the agent knowledge (the decision-making process) from its concrete functionalities (capabilities execution). Such functionalities can be modelled using TROPOS plan decomposition diagrams and following quite standard object-oriented methods (e.g. using UML 2.0 activity- and sequence-diagrams). An automatic mapping from these diagrams to an implementation within a software agent based on the platform Jade is proposed in [Penserini et al., 2006b].

This thesis describes a way to effectively use the knowledge level information, modelled by TROPOS diagrams, to implement the reasoning part of an agent. That is, goals hierarchies (goal decompositions), goal dependencies, contribution links towards quality of services (by softgoal concepts), and means-ends relationships have to be mapped to code according to design time semantics, to characterize both the agent perception about the environment and its social responsibilities. So, if an agent is aware of this knowledge, in order to achieve goals, it can select the subgoals to pursue and the best functionalities to execute, according to the actual environment, context conditions, contributions to other goals, and failed intentions, monitoring failures and managing failure recovery.

These demands naturally accommodate with an implementation as “rational agent”, using a BDI architecture (Belief-Desire-Intention, [Rao and Georgeff, 1991]). In BDI-based Agent Platforms (see Section 2.1.1), the environment is modelled by a set of beliefs, goals to achieve are desires and intentions can be realized executing plans or, respectively, the delivered capabilities. In particular, this proposal adopts Jadex [Pokahr et al., 2005], which provides full support for BDI agents and is FIPA-compliant.

The proposed approach defines a direct and automatic mapping of goals, goal dependencies and goal decompositions from the TROPOS Architectural Design phase to the code, to correlate knowledge level information with runtime instances of software agents. A diminished set of TROPOS structures was figured out for the Detailed Design phase and concrete semantics were associate to it in order to be properly able to
characterize an agent framework, with focus on the BDI-like agent-oriented concepts. Notice that, the defined agent framework aims more to be flexible enough to support the above requirements than to possibly embed some more domain specific AI techniques such as Planning or Machine Learning.

Basing on this mapping, a prototype for an automatic generation of agent-oriented code from TROPOS diagrams is implemented, to extend the design tool TAOM4E by CASE-features. The code generation tool, called \textit{t2x}, delivers an executable Jadex-based agent structure, strongly reducing the effort for the implementation of the modelled concepts.

\section*{1.3 Thesis Outline}

The following Chapter 2 gives an outline of the Agent Oriented concepts used in this work, presents the most popular and interesting AOSE methodologies, and presents the used agent platforms. In Chapter 3, a methodological approach is given, showing important characteristics of the used methodology, investigating the need for bringing the knowledge level to the implementation, and giving precise semantics for knowledge-related Tropos concepts. Chapter 4 defines the mapping from a set of modelling concepts to the implementation and outlines the translation procedure, while Chapter 5 shows one of the preliminary experiments with the implemented code generation tool, made to test the applicability of the approach. Finally, Chapter 6 summarizes the results obtained and gives an outlook to the future work to be done.
2 Agents - State of the Art

This section gives a brief overview on the concepts used in the thesis. Some definitions from literature are given for the term “Software Agent”. Various frameworks extend traditional languages to support agent-oriented programming and include platforms upon them the implemented agents can be executed. Furthermore, the huge conceptual differences to objects justify the need for agent-oriented software engineering methodologies; some of them are founded on object-oriented methodologies, others on new approaches conceived to better catch the nature of agents. So called “Reasoning Agents” or “Intelligent Agents” can be built following the BDI-model. Several methodologies and platforms support or base on this further abstraction.

2.1 Software Agents

“If a problem domain is particularly complex, large, or unpredictable, then the only way it can reasonably be addressed is to develop a number of functionally specific and (nearly) modular components (agents) that are specialized at solving a particular problem aspect.” This definition, from [Sydorac, 1998], gives a very general and basic reason for the Agent Oriented Programming paradigm (AOP), modularity and decomposition. But Agents should be more than that. They are a new conceptual paradigm for analysing stakeholders' requirements and for designing, and implementing complex, distributed, and interactive systems. Systems built on the concept of Agent with its Behaviours, can be conceived from a much more natural and social point of view, than as an architecture of objects and methods.

To be called an Agent, a software entity must have some properties that delimit it from simple objects and leads to a definition that resembles as much as possible to the “traditional” conception of the term Agent. Van Steen and Tanenbaum [Van Steen and Tanenbaum, 2002] give a general definition that catches the main features of a Software Agent:

“We define a software agent as an autonomous process capable of reacting to, and initiating changes in, its environment possibly in collaboration with users and other agents. The feature that makes an agent more than just a process is its capability to act on its own, and, in particular, to take initiative where appropriate.”
This definition can be further restricted adding several properties, like sociability, rationality, mobility, and abilities such as reasoning, cooperation, negotiation, learning,... Details for some important properties can be found in [Elam, 2005][Wooldridge, 2002].

For the scope of this thesis, Agent can be seen as autonomous, reactive or proactive software entities, which can interact and cooperate to reach their goal. This work, and agent-oriented software engineering in general, aims more at building complex, distributed and possibly dynamic agent systems, so for the used agents it is not important if they are intelligent or not, or if they are endowed with particular planning or reasoning algorithms. Indeed, “it is often the case that multiple, non-complex agents may interact in such as way that the entire system may exhibit seemingly intelligent behaviour” [De-Loach, 2001].

**Differences to OOP**

Often, the concept of Agent is seen as an evolution of the Object Oriented concept. Like OOP, AOP founds on the principles of decomposition, information hiding, and interaction. However, there are important conceptual differences between them.

Objects encapsulate data and methods to control it, however they have no control on when these methods are called to be executed from outside. Agents *react* to requests for an activity or for the satisfaction of a goal. The decision on what actions to execute and on how to respond remains in their responsibility. You do not call an agent's activity, but you ask for satisfying a request. “*Objects do it for free; agents do it because they want to*” [Wooldridge, 2002] – this concept underlines the social view to programming promoted by AOP. Furthermore, in opposition to objects, agents are always active and can exhibit a goal-directed behaviour by taking the initiative, performing activities and sending requests to others [Wooldridge and Ciancarini, 2001].

**“Intelligent” or “rational” Agents**

Agents are often endowed with some “intelligent” capabilities, e.g., they have goals that they try to achieve, they may have a knowledge base in terms of an ontological domain representation, reason over its knowledge base in order to make decisions, interpret and execute partners’ requests, and the like [Penserini et al., 2006c]. According to Wooldridge and Jennings [Wooldridge and Jennings, 1995], agents should have the following properties and skills to be called “intelligent”: 
2.1 Software Agents

- Reactivity: to sense/perceive the environment and consequently react to possible environment stimulus/changes to satisfy its design objectives.

- Proactiveness: to autonomously decide the more convenient behaviour to achieve its design objectives, namely they exhibit a goal-direct behaviour.

- Social ability: to interact with other partners (human being and software agents) to satisfy their design objectives.

As the qualification “intelligent” for agents is collocated more in the ambit of artificial intelligence, we would prefer the term “rational agent” to define the Agents used in the software engineering community and in this thesis. The definition found in [Wikipedia: Rational agent] suites best to this work:

A rational agent takes actions which, given the knowledge of its environment, maximizes its chances of success. The action a rational agent takes depends on the agent’s past experiences, the agent’s information of his environment, the actions available to the agent and the estimated benefits and chances of success of that action.

A well-known model that represents rational agents within mentalistic concepts is the BDI-architecture.

2.1.1 The BDI model

The BDI-model [Rao and Georgeff, 1991], with roots in the theory of practical reasoning by Bratman [Bratman, 1987], bases on the three mental state concepts of Belief, Desire, and Intention. Beliefs are the agent’s knowledge about the current state of the environment; it may be incorrect or incomplete. Desires are the states of affairs, that the agent’s would, in an ideal world, wish to achieve. They should be consistent with one another. Intentions are the activities, which the agent has committed for execution, to satisfy a desire.

The BDI-theory has been rigorously formalized in a family of BDI-logics, (e.g. BDICTL, Logic of Rational Agents [Wooldridge, 2000]). These logics group a first order logic, the temporal logic CTL* and the concepts of belief, desire and intention, to express relationships between the agent’s mental states and the rational actions to be performed in a particular state.

The BDI-model naturally fits to agents that are goal directed, communicate with others and can react to changes of the environment they are embedded in.
For an executable model, Rao and Georgeff introduce the more concrete concepts of *Goals* [Rao and Georgeff, 1991] and *Plans* [Rao and Georgeff, 1995]. Goals are chosen, consistent desires an agent tries to fulfil (believing that they are achievable), plans contain the means of achieving certain future world states (the possible actions to execute in order to reach a goal), where intentions are a set of plans the system tries to execute to reach a goal. Most concrete BDI architectures use a model based on the three concepts *belief*, *goal* and *plan*.

Some authors extended this model, including *perceptions or events*. Taveter and Wagner [Taveter and Wagner, 2005] argue that perceptions are an important concept to model communication, but cannot be conceptually treated as beliefs (as changes in the environment), because they are transient, while beliefs are persistent.

Another concept introduced for rational agents is the *Capability*. Although there exist different definitions of this term in the agent-oriented community, they have in common that capabilities add modularity to the inner architecture of an agent, grouping plans (the “ability”), goals and related concepts.

The BDI architecture has been implemented and successfully used in a number of complex applications. Moreover, several AOSE methodologies and agent platforms are based on, or support parts, of this architecture.

### 2.1.2 Multi-Agent Systems

Multi-agent systems (MAS) are systems composed of several autonomous agents that can work collectively to reach individual or common goals that are difficult to achieve by an individual agent or a monolithic system, following the idea of a human organisation. Complex or distributed systems can be modelled as MAS to achieve a better decomposition of the problem and to have a more natural way to model societies of collaborative entities. A MAS could also include human actors and service providing actors like web-server or databases, which are not necessarily agent-oriented applications.

Each single agent in a multi-agent system is autonomous and without centralized control and can communicate through message passing (communicating systems) or through reciprocal chances to the environment (situated systems). Multiple agents could depend one from another for a specific activity, they could delegate goals and activities to others, and they could share their environmental knowledge and even some of their
abilities. If we use a methodology that supports the definition of roles, multiple agents could play one role and different roles could be played by a single agent.

The Agent Oriented Software Engineering Methodologies, presented in the next section, use different concepts to capture the actors of the modelled system and the social relationships and interaction protocols between them, to develop a multi-agent system.

2.2 AOSE Methodologies

The design of agent-oriented systems needs to be backed by an appropriate software engineering methodology that supports the new concepts and the new view to software systems propagated by this new programming paradigm. The engineer has to be guided through the software development phases, from the requirements analysis to the implementation, with an agent-oriented system in mind, to be able to use all its benefits.

Traditional object-oriented methodologies are not suited to develop agent-oriented systems, not only because they lack the concept of Agent, but also because systems, with their organisational structures, are modelled from an architectural, not from a social point of view. In particular, an agent's flexible, autonomous goal-oriented behaviour and the complexity of the dependencies and interactions cannot be adequately captured by such approaches. Moreover, Wooldridge and Jennings [Wooldridge and Jennings, 1998] identify the major pitfalls in designing an agent-oriented system and underline the need for a systematic framework that addresses the pragmatic, engineering aspects of agent-based systems.

All these considerations lead to the development of various Agent Oriented Software Engineering Methodologies, some of them are shortly presented along next pages. Further details and additional methodologies can be found in [Henderson-Sellers and Giorgini, 2005]. MaSE, Gaia and PASSI extend existing object-oriented approaches, Prometheus and Hermes were built on the experience with a concrete agent platform and MAS-CommonKADS has its roots in the Knowledge Engineering. The work in this thesis bases on TROPOS, a methodology, which was influenced by Eric Yu's organisational modelling framework * [Yu, 1995] and therefore by requirements engineering concepts. TROPOS is presented, with all the details needed for this thesis proposal, in Section 2.4.
2.2.1 MaSE

The Multi-agent Systems Engineering methodology (MaSE) [Wood and DeLoach, 2000] was originally designed to develop general purpose, closed (with a-priori known number and types of agents), and heterogeneous multi-agent systems and has been used for the development of various systems from heterogeneous database systems to cooperative robotics systems. MaSE builds on existing object-oriented methodologies and uses UML diagrams with some extensions, see [DeLoach and Kumar, 2005] for an overview. The relationships between agents and their functionalities cannot be dynamic and interactions can be only pair-wise.

2.2.1.1 The development phases

The methodology has seven development steps, depicted in Figure 2.1 that can be used iteratively adding more information with each pass, until finally a complete system design is produced. The system analysis process is captured in three steps: capturing goals, applying use cases, and refining roles. In the design phase the roles and tasks de-
2.2 AOSE Methodologies

developed in the analysis phase are converted to agents and conversations, two concepts that are more appropriate for an implementation.

**Capturing goals:** In the first step you extract the goals from the system requirements. From the initial set of requirements (that can exist in an informal text form), the scenarios and then the high-level goals and subgoals of the system are extracted, analysing “what” has to be done. These goals are then categorized into four types (summary goals, non-functional goals, combined goals, and partitioned goals) and structured into a goal tree, represented as Goal Hierarchy Diagram (Figure 2.2 a). This is the basis for the analysis of the entire system and the definition of the system roles. The approach is similar to the KAOS approach [Dardenne et al., 1993], except that MaSE goals are not strictly AND- or OR-refined.

**Applying Use Cases:** In the second step all goals and subgoals are translated into use cases to represent the desired behaviours of the system. The captured use cases are restructured as a sequence diagrams (Figure 2.2 b, similar to the UML sequence diagrams), that depict sequences of messages between multiple agent roles.

---

**Figure 2.2:** Sample MaSE diagrams for the Architectural Design phase (from [Wood and DeLoach, 2000]).
Refining Roles: MaSE is built on the assumption that system goals will be satisfied if each goal maps to a role and each role is played by an agent. In this last step of the system analysis phase, the identified goals are refined and mapped typically one to one to roles. This step is captured in a Role Model (Figure 2.2 c) that represents roles (rectangles) and decomposes them in a set of tasks (ovals). Solid lines represent external role-to-role communication, while the internal communication between tasks in a role is indicated with dashed lines. To define the behaviour required for a specific role, all its tasks are detailed, using a Concurrent Task Diagram (Figure 2.2 d), based on finite state automata.

Construction of Agent Classes: The first step of the design phase maps the roles identified in the previous phase to agent classes, which is documented in an Agent Class Diagram (Figure 2.3 e). These diagrams are similar to object-oriented class diagrams: Agent classes are defined by the roles they play (instead of their attributes and methods) and the relationships between agents are always communications. Each role is mapped to at least one agent class; if multiple roles are mapped to the same agent class, they may be performed either sequentially or concurrently. The constraint, that each goal is mapped to a role and each role to an agent, ensures that all goals are actually implemented in the system.
2.2 AOSE Methodologies

**Constructing Conversations:** Basing on the communication protocols identified in the analysis phase, the detailed design of the conversations is modelled in special Communication Class Diagrams (Figure 2.3 f). These diagrams are based on finite state automata and use states and transitions to define the inter-agent communication. The conversation protocols have to be made robust and to take into consideration failures of the participants.

**Assembling Agent Classes:** In this step the agents’ architecture and their individual components are defined. Here the developer has to decide if to use a BDI-based or some other architecture. Agent Architecture Diagrams (Figure 2.3 g) capture these decisions.

**System Design:** The last step of the MaSE methodology consists in drawing UML-like Deployment Diagrams (Figure 2.3 h) to show the numbers, types, and locations of agent instances within a system.

The methodology is supported by the graphically based agentTool development system. With agentTool it is possible to draw the diagrams for the MaSE analysis and design phases and then to apply a semi-automated code-generation process for the frameworks Carolina and AgentMom.

### 2.2.2 Gaia

Gaia, developed by Wooldridge, Jennings and Kinny [Wooldridge et al., 2000], is a methodology for the analysis and design of single agents as well as multi-agent systems. It deals with macro-level (societal) and micro-level (agent) aspects of a system. Early requirements engineering and the implementation phase are not covered by this methodology. In Gaia, a MAS is viewed as a computational organization, with a static organizational structure, that consists of various interacting roles. The goal of the overall system should be to maximize some global quality measure. The developers state that agents should be coarse-grained computational systems, each one needing resources like a typical UNIX process.
2.2.2.1 The development phases

Gaia defines a systematic process to go from the requirements to a sufficiently detailed design that can be implemented directly, moving from abstract to increasingly concrete concepts. Analysis and design can be thought of as a process of developing increasingly detailed models of the system to be constructed. The process consists of five models for the analysis and design phase, whose relationships are detailed in Figure 2.4.

![Figure 2.4: Relationships between Gaia models (from Wooldridge et al., 2000)](image)

**Analysis:** The first step is to capture the system organization as a collection of roles with certain relationships. The roles are defined by four attributes: responsibilities, permissions, activities, and protocols. The responsibilities contain liveness- and safety properties, defined in a formal language, permissions are the rights associated with a role. The activities of a role are the computations associated with the role, the “actions” that can be executed. Finally, protocols define the ways to interact with other roles. They can complex, e.g. an auction type, a negotiation protocol, etc. The roles are captured in the role model, the interaction model defines the protocols, describing participant roles, purpose, inputs and outputs and the computations to perform during the interaction.

**Design:** The design process does not aim at transforming the abstract models got from the analysis to less abstracted models that are easier to implement. Rather, the aim in Gaia is to transform the analysis models into a sufficiently low level of abstraction that traditional design techniques, including object-oriented techniques, can be applied to implement agents. The following three models are used:

- The agent model describes the different agent types (sets of agent roles) and the run-time instances of them.
2.2 AOSE Methodologies

- The services model identifies the services associated with each role and specifies its main properties. Services are functions of the agent, with defined inputs, outputs, pre- and post-conditions. Their implementation could be realized in any framework, or decomposed into object-oriented methods.

- The acquaintance model is a directed graph that defines the communication links between the agent types, derived in a straightforward way from the roles, protocols, and agent models. It should help to identify communication bottlenecks and to ensure that the agents are loosely coupled. If problems arise, the analysis and design models have to be revisited.

For the Gaia models only the contents and procedures, but no special-purpose notation was defined. Developers are suggested to adopt simple and intuitive notations, but the possibility to use AUML or temporal logics is not excluded. Some extensions to Gaia were proposed to improve the modelling of the environment and the social dependencies between agents.

2.2.3 Passi

Passi (a Process for Agent Societies Specification and Implementation), proposed by M. Cossentino and C. Potts [Cossentino and Potts, 2002][Cossentino, 2005] is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies integrating design models and concepts from both OO software engineering and artificial intelligence approaches\(^1\). It was conceived using standards, whenever possible, therefore all diagrams use the UML modelling language, with the extension mechanisms (constraints, tagged values and stereotypes), used to represent agent-oriented concepts. Passi is supported by the Passi ToolKit (PTK), an add-in for IBM Rational Rose, plus AgentFactory, a tool for reusing agent patterns and generating code. The methodology consists of five models, concerning the different design levels, with a total of twelve processing steps.

\(^1\) cited from http://mozart.csai.unipa.it/passi/passi.htm, where further information on Passi can be found.
2.2.3.1 Models and development phases

**System Requirements Model:** This step consists of building a model of the system requirements with social characteristics, in terms of agency and purpose. First, using conventional use-case diagrams, a functional description of the system is made. In these diagrams, one or more use-cases are grouped into a package with stereotype `<Agent>`, to define the responsibilities of the single agents. Then, these responsibilities are elaborated, through role-specific scenarios, described in sequence diagrams. Moreover, the capabilities of each agent are specified, using activity diagrams.

**Agent Society Model:** Here the dependencies and the social interactions among the agents discovered in the first model are defined. With class diagrams, the ontology for the agents' knowledge and communication is defined. Furthermore, class diagrams are used to represent the roles played by the agents, the tasks involved in these roles, communication capabilities and inter-agent dependencies. The grammar of the protocols is described in UML sequence diagrams.

**Agent Implementation Model:** The solution architecture is modelled in terms of classes and methods, following these two steps: First, the structure of the agent classes is specified in conventional class diagrams. Then, activity diagrams or state charts are used to describe the behaviour of individual agents. These diagrams are refined through an iterative process. The result of this stage is to obtain a detailed (typically platform dependent) structure of the software, ready to be implemented almost automatically.

**Code Model:** In this step the solution is coded. The PTK tool can be used to generate a skeleton from the models. Reusable parts of the methods' implementation, taken from a patterns and design descriptions library, are added by the AgentFactory tool. The plat-
forms used for the implementation (Jade, FIPA-OS) are FIPA-compliant and can therefore use a number of specialized interaction patterns. Then, the source code can be completed manually, following the structures modelled in the diagrams.

**Deployment Model:** These models the distribution of parts of the system across hardware processing units, and their migration between processing units, illustrating the allocation of agents to processing units and their movements in a UML deployment diagram with extended syntax for mobility (dashed lines with stereotype `move_to`).

Recently, a **testing phase** was added: with single-agent tests, the behaviour of each agent concerning the original requirements of the system is controlled. Society tests are carried out after system deployment and to verify the integration.

The methodology has been used in several research projects and in a university course at the University of Palermo.

### 2.2.4 MAS-CommonKADS

The methodology MAS-CommonKADS, first described in [Iglesias et al., 1998], integrates techniques from the *knowledge engineering* methodology CommonKADS [Schreiber et al., 1994], which can be seen as a standard for knowledge modelling, and extends it for multi-agent systems (MAS) modelling, with techniques from object-oriented and protocol engineering methodologies.

The methodology defines conceptualization, analysis and design phases and defines seven development models for analysis and design of the system. MAS-CommonKADS uses mainly UML 2.0 diagrams and is supported by a Java CASE tool and a Plug-In for IBM Rational Rose.

#### 2.2.4.1 Models and development phases

**Conceptualization phase**

During this phase the requirements elicitation is carried out, to obtain a preliminary description of the problem, drawing use-case diagrams (extended to distinguish human agents, with a round head, and software agents, with a square head) and defining the interactions (complete with alternatives, iterations, exceptions, etc.) in message sequence charts. These interactions will be refined in the next models, defining the knowledge interchanged.
**Analysis phase**

This phase provides full requirements specification of the multi-agent system, through the development of five models:

The **Agent model (AM)** identifies and describes agents and acts as a link between the rest of the MAS-CommonKADS models, that specify agent characteristics: reasoning capabilities, skills (sensors/effectors), services, agent groups and hierarchies. Different strategies are proposed to identify the agents. Every agent should be described using textual templates that collect its main characteristics: name, type, role, position, offered services, goals, skills, permissions, etc.

The **Task model (TM)** describes all the activities (called tasks) that the agents can carry out in order to achieve a goal. It is defined by its name, its goal, a short description, input and output, preconditions, the required capabilities of the performers, etc. Tasks are decomposed following a top-down approach, in an AND/OR-tree. Alternatively, UML activity diagrams can be used. If the tasks defined are particularly knowledge intensive or involve interactions with agents or humans, they can be further developed, respectively, in the expertise-, coordination- or communication model.

The **Coordination model (CoM)** describes the conversations between agents to fulfil a goal in a cooperative way. Interactions, protocols, and required capabilities have to be modelled by flow diagrams, message sequence charts, and state transition diagrams.

The **Expertise model (EM)** describes the knowledge and the reasoning capabilities needed by agents to carry out their tasks and achieve their goals, using the main parts of the CommonKADS framework. Four types of knowledge should be modelled: domain knowledge, inference knowledge, task knowledge and problem solving knowledge.

The domain knowledge represents the declarative knowledge of the problem, modelled as concepts, properties, expressions, and relationships using the Conceptual Modelling Language (CML) or the graphical notation of the Object Model of OMT. Inference knowledge represents the inference steps performed for solving a task, mainly adapting generic inference types from a library, task knowledge describes the order of the inference structures. Then, problem-solving methods are defined for each inference type. This knowledge modelling framework has been successfully applied in several projects and provides languages, tools, and a rich library of components.
2.2 AOSE Methodologies

The **Organisation model (OM)** describes the organisation into which the knowledge-based MAS is going to be introduced and the social organisation of the agent society. The notation is based on UML class diagrams, extended with an agent structure with mental states and external interfaces and additional semantics for agent-to-agent associations: inheritance and grouping (Figure 2.6).

The **Communication model (CM)** details human-software agent interactions and human factors for developing user interfaces.

![Figure 2.6: Organizational model class diagram notation (as in [Iglesias and Garijo, 2005]).](image)

**Design phase**

The **Design model (DM)** collects the information retrieved in the analysis phase and details how requirements can be achieved. It consists of three sub-models: The *network design* for designing the relevant aspects of the agent network infrastructure (required network and knowledge facilities), the *agent design* for dividing or composing the agents of the analysis, according to pragmatic criteria and selecting the most suitable agent architecture for each agent, and the *platform design* for selecting the agent development platform for each agent architecture. All this data is filled only into a textual template, no diagrams are defined.

This methodology gives most weight to accurate requirements engineering and knowledge modelling, but only limited support for a straight-forward development process from modelled requirements and knowledge to design and implementation phases. MAS-CommonKADS has been applied in several research projects, from an intelligent network management to a simulation of stakeholders.
2.2.5 Prometheus

The Prometheus methodology [Padgham and Winikoff, 2002][Padgham and Winikoff, 2005] was developed at the RMIT, Australia, in collaboration with the company Agent Oriented Software Pty. Ltd., the developers of the commercial system JACK, which contains a proprietary agent platform and tools for agent-oriented programming. Prometheus is a pragmatic methodology that was developed following the software engineering experiences on this concrete platform, but besides the detailed design, the methodology remains general. Prometheus defines agent-oriented concepts, notations, and a development process with guidelines and techniques that define how to carry out this process.

In Prometheus, an agent has beliefs and plans. Precepts are the information retrieved from the environment; agent actions can affect the environment. Proactive agents pursue goals, while reactive agents respond to events. Finally, agents are social and use messages to communicate. These messages are collected in interaction protocols. This set of concepts is enough to design systems with a BDI-architecture.

2.2.5.1 The development phases

Prometheus consists of an iterative process with three development phases: System Specification, Architectural Design and Detailed Design (Figure 2.7). The diagrams that are defined are not based on UML, leaving out its object-centric viewpoint, but, at the same time, renouncing to its familiar and tool-supported notation.

System Specification phase

The system is specified in terms of system goals, use case scenarios, functionalities and the interface to the environment; no particular order is given to these activities. An initial set of goals is extracted from the system description. Then these goals are developed into more refined subgoals, asking “how” they could be achieved [van Lamsweerde, 2001]. The goals and subgoals are represented in a goal diagram, a simple DAG that connects goals (depicted as ovals). A system functionality consist of the related goals and perceptions, the actions that it performs, the data it uses, and the triggers (percepts and events) that activate it. Functionalities are described using descriptors, simple textual forms filled with all the relevant data.

Another activity in the specification phase is the definition of use-case scenarios. Scenarios are textual descriptions of a particular example sequence of events performed
2.2 AOSE Methodologies

to reach a goal or to respond to an event and include also variations in the proceedings. Finally, the actions the system will be able to perform, and the processing of the percepts, are specified.

**Architectural Design phase**

In this phase, the architecture of the system is specified, drawing several diagrams. Functionalities are explored using a coupling diagram and an agent acquaintance diagram. Basing on their coupling, the functionalities are grouped into *agent types*. Interaction diagrams are derived from the use-case scenarios, and then generalized to interaction protocols.

At last, the overall system structure is specified and documented in a system overview diagram (Figure 2.8). This diagram collects information from the various descriptors and shows agents, percepts, actions, messages, and data as nodes. Directed arcs indicate messages being sent and received, actions being performed, perceptions being received and data being read and written by agents.

*Figure 2.7.: Overview on the Prometheus development phases and artefacts (from [Sudeikat, 2004]).*
Detailed Design phase

Basing on the specifications, the internal structure of a single agent type is designed and all details, which are important for the implementation, are described. All agent capabilities are defined in capability descriptors and then decomposed to sub-capabilities and plans, using capability diagrams. In event descriptors, plan descriptors, and data descriptors the relative elements are detailed. For example, a plan descriptor contains triggers (the events that cause the execution of the plan), context conditions, and the body of the plan. The agent overview diagram (Figure 2.9) gives an overview on all capabilities and the agent-internal message- and data-flow. Together with the descriptors it gives a precise picture of the system to implement.

![Figure 2.8: Prometheus system overview diagram (from Sudeikat, 2004).](image-url)
2.2.5.2 **Tool support and final considerations**

Tool support is vital for developing large systems, especially in iterative methodologies such as Prometheus, to guide the designer, to propagate changes and to avoid inconsistencies across the different models. The freely available Prometheus Design Tool (PDT) supports developers in creating consistent Prometheus diagrams and a report that contains the complete design. Further functionalities are under development [Padgham and Winikoff, 2005].

![Prometheus agent overview diagram](image.png)

*Figure 2.9.: Prometheus agent overview diagram (from [Suidekat, 2004]).*

Jan Suidekat [Suidekat, 2004] developed a tool that generates a Jadex Agent Definition File (see Section 2.3.2) from a PDT design. The process has to be assisted by the developer through an interactive graphical interface. The commercial platform JACK includes the Jack Development Environment (JDE). JDE provides a graphical interface to model the diagrams of the detailed design phase. From these diagrams, JACK code skeletons can be generated. Moreover, the tool ensures the consistency of the code with the diagrams, reflecting changes of the code to the diagrams and vice-versa.

Prometheus has been used in some research projects and university courses. Its developers claim that with this methodology, non-graduate people successfully implemented reasonable agent systems. On the other side, Prometheus gives not many support to the early requirements and business process analysis and does not deal with mobile agents.
2.3 Agent Platforms

Although today many research projects and some industrial systems are based on AOP, no agent-oriented languages are used in practice. However, many tools have been developed to support implementation and execution of multi-agent systems. They are mainly based on Java and try to abstract from the object-oriented paradigm, introducing agent-oriented extensions and agent platforms. Agent platforms are middleware that delivers basic services for the realization of multi-agent systems, such as communication protocols and special services. Several agent platforms are available, the best known of them are JACK, Fipa-OS, Zeus and Jade.

The IEEE-Foundation for Intelligent Physical Agents\(^1\) (FIPA) defines a standard for messages and platform architecture of an agent-oriented system, to facilitate the interoperability of independently developed agent systems. The FIPA standard defines services that have to be provided by the platform, such as agent management service (AMS) and directory facilitator (DF), a message format, a language (Agent Communication Language, ACL), communication protocols between agents and the agent life cycle phases. The architecture enables agents to communicate through a standardized message transport system, also called Agent Communication Channel (ACC), which is the software component controlling all the exchange of messages within the platform.

The two platforms important for this work are now presented: The Jade platform, fully FIPA-compliant and widely used, and Jadex, a Jade-based agent environment that allows developing goal-oriented agents following the BDI-architecture.

\(^{1}\) An IEEE standards organization. The Foundation for Intelligent Physical Agent (FIPA) makes available its specification at: http://www.fipa.org/.
2.3 Agent Platforms

2.3.1 Jade

Jade\(^1\) (Java Agent DEvelopment Framework) is an AOP framework developed by Telecom Italia Lab (TI Lab). It bases on Java and allows the implementation of multi agent systems through middleware that complies with the FIPA specifications, Java classes to be extended to program agents, and a set of graphical tools that supports the debugging and deployment phases. The agent platform can be distributed across heterogeneous machines and supports mobile agents that can move from one system to another on the distributed platform. The communication architecture offers flexible and efficient messaging, where Jade creates and manages a queue of incoming ACL messages, private to each agent. The full FIPA communication model has been implemented: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies, and transport protocols. The message transport mechanism adapts to each situation by transparently choosing the best and fastest available protocol: Java RMI, event-notification, or IIOP are used, depending on the location of the involved agents. Jade has been made freely available under LPGL v.2.

Agents in Jade

In Jade, user-defined agents are implemented, in Java, as an extension of the Jade Agent class. This implies the inheritance of features to accomplish basic interactions with the agent platform (registration, configuration, remote management, etc.) and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use of standard interaction protocols, etc.). Agents have an initialization method (setup), which is the point where any application-defined agent activity starts and the first behaviour has to be scheduled. Behaviours, contained in Behaviour classes, are the activities an agent is able to execute, and can be scheduled for execution in an agent-internal multitasking environment. Behaviours can be scheduled for one-time or cyclic execution.

The implemented Java code has to be launched through the jade.Boot class, that boots the Jade platform (or connects to an already running platform) and adds the agent, registering it with the AMS and starting, for each agent, a simple non-preemptive round-robin scheduler, which switches between the scheduled behaviours. Communications between agents can be built using provided send- and receive-methods or ready-

\(^1\) [www.jade.tilab.com](http://www.jade.tilab.com)
made behaviour classes that implement FIPA-standard interaction protocols, such as FIPA-Request, FIPA-query, FIPA-subscribe and FIPA-Contract-Net.

**Platform architecture**

A Jade platform consists of at least one *Container*, a running instance of the Jade runtime environment that can contain several agents.

To comply with FIPA specifications, the Jade platform has to provide at least two standard services: The Agent Management Service (AMS) and the Directory Facilitator (DF). They are realized as agents and are automatically executed at platform initialization, in the main container.

The DF implements a standard yellow pages service. Agents can register their services at the DF and they can be searched through the DF, through a simple, message-based interface. The AMS provides white page services and has supervisory control over access and use of the agent platform. All agents automatically register to it whereby they get associated to a unique ID. Agents can be created and killed by the AMS.

**Tools**

The Jade platform is provided with a graphical user interface, where the agent life cycle can be managed, communications can be tracked and messages can be sent by dummy agents for testing purposes. In particular, the *Sniffer Agent* (Figure 2.11) can be started from the main interface. It displays selected agents and all messages sent between them. The message performative is signed and message contents can be observed. The Sniffer is a helpful accessory for debugging, to follow the execution of complex interaction protocols.
2.3 Agent Platforms

2.3.2 Jadex

The Jadex agent framework\(^1\) [Pokahr et al., 2003][Pokahr et al., 2005] is built on top of the Jade platform and allows agent developers to implement Jade agents, which exhibit a rational, goal-oriented (opposed to task-oriented) behaviour, by implementing a BDI architecture. Jadex consists of an API, an agent platform and development tools.

\(^1\) Jadex is available under GNU LGPL license, at http://source-forge.net/projects/jadex.
An agent is programmed defining an XML file, called *agent definition file* (ADF), and Java classes. In the ADF, the agent is specified by declaring *beliefs*, *goals*, available *plans* and some other properties. In addition to this, for each plan used by the agent the plan body has to be implemented in a separate Java class.

**Beliefs**

The *beliefs* section defines the content of an object-oriented database. Jadex can store single beliefs and sets of beliefs, consisting of Java objects. Beliefs can be queried by using simple OQL-like SELECT-statements, that can be predefined in the ADF’s *expressions* section. Beliefs can be accessed in all conditions for goals and plans and in the plan body files.

**Plans**

Plans contain actions that can be executed to achieve a goal. In the ADF *plans* section, a plan definition contains references to the goals and messages that can trigger its execution, pre- and context conditions for the execution, and the associated plan body, implemented as ordinary Java class. This class has to extend a Jadex *Plan* class and implements all the needed functionalities as conventional Java code. Using the Jadex API, the plan body may access the belief base and deliver sub-goals. Other plans are executed to achieve these sub-goals, thereby forming a hierarchy of plans. The Jadex agent keeps track of the actions and sub-goals carried out by each plan, to determine and manage plan failures. Out of the triggered plans for some event, only those are set as executable (*or applicable*), which have no invalid conditions and have not yet failed in the current goal achievement process (this is the default behaviour, that can be customized).

**Goals**

The goal is a central concept, representing a certain target state that the agent is trying to reach. In a goal-oriented design like Jadex, goals explicitly represent the states to be achieved, and therefore the reasons, why actions (plans) are executed. In contrast to other BDI systems such as JACK or Jason, goals are not only transient events, but an agent is aware of its goals at any time [Pokahr et al., 2005b]. Goals are explicitly represented and have a life-cycle with different states.

The *goals* section contains all the goals defined for an agent. Goals can have different types and different conditions for goal activation and goal achievement. Different
kinds of goals can be distinguished: achieve-, query-, perform- and maintain-goals. The most interesting are achieve-goals and maintain-goals. Achieve-goals can be defined with or without a target state. With target state, they try to execute one of the applicable plans until this state is reached. Otherwise, they are achieved with the first plan executed successfully. Maintain-goals are similar, but they continuously execute appropriate plans to re-establish the target state whenever needed.

There are two distinct activities a Jadex agent performs in order to reach its goals: Deliberation and Meta-level reasoning. Deliberation deals with deciding what goals out of the currently adopted ones should be actually active. Jadex implements a simple strategy, which checks context- and achievement conditions and possible inhibition links between goals to decide, which goals to activate.

Meta-level reasoning deals with deciding what plan to pursue to achieve a goal. If more than one plan is applicable for an active goal, a meta-reasoning process starts: a so-called metagoal is dispatched, which triggers an associated plan, the metaplan, that im-
implements a strategy to select between applicable plans. The default strategy, that uses plan priorities or follows the definition order in the ADF, can be easily extended by the developer (Figure 2.14). Meta-reasoning is executed until one plan succeeds, but only if more than one plan, excluding failed ones, is applicable for that goal.

**Tool support**

As a Jadex agent is still a Jade agent all available tools of Jade can also be used to develop Jadex agents. Jadex agents can be executed through the graphical Control Center, on an Jade platform extended by the Jadex Remote Monitoring Agent. The Jadex Introspector plug-in can be used to observe internal state of agents including their beliefs, goals and plans. The Tracer Agent can be used to visualize the internal goal achieving processes of an agent at runtime and show causal dependencies among agent's beliefs, goal, plans and message events, textually or in a graphical, molecule-like fashion. Among the development tools, Jadexdoc is worth to be mentioned. This documentation tool creates JavaDoc-like documentation from ADF files.

### 2.4 TROPOS

The agent-oriented software development methodology TROPOS [Castro et al., 2002] [Bresciani et al., 2004] was developed as a joint work between the Universities of Toronto (Canada), Louvain (Belgium), Pernambuco (Brazil), and Trento (Italy) and ITC-Irst (Italy).

TROPOS covers all development phases from the requirements analysis to the detailed design and gives a crucial role to the very early requirements analysis, adopting the i* modelling framework [Yu, 1995] for agent-oriented programming. The concept of Agent and related mentalistic notions such as goals, plans and dependencies are used through all the development phases, starting from the modelling of stakeholders and their intentions and leading to an architectural design of a multi-agent system. Having agent-oriented programming in mind, the principal aim is to conceive design and implementation as a program at a knowledge level, avoiding a translation of the concepts to traditional software level notions such as classes and methods of object-oriented programming. Using the same concepts in all phases, another benefit would be the ability to track the decisions made in one phase through all the development process.
2.4 TROPOS

2.4.1 Key concepts

The Tropos models rely on a conceptual metamodel that contains all entities and relationships usable within the methodology.

The **Actor** is the key entity and represents a human, social or software *agent*, a *role* or a *position*. Actors can contain **goals** they try to achieve, **plans** they are able to execute to affect the environment and stored **resources**. **Roles** characterise behaviours within a specialized domain, such as “buyer” or “seller”, and can be played by an agent. Agents can occupy different **Positions**, which contain groups of roles, for example “tradesman”, “student” or “teacher”.

**Goals** and **Plans** represent, like in a BDI architecture, an actor's strategic interest and a way of doing something. There exist two types of goals: **Hard goals** are functional requirements and have a measurable satisfaction condition, a value or a threshold that can be reached. For **Softgoals**, no precise criteria for its satisfaction can be found,
typically they model non-functional requirements such as quality of service or security properties. “Be happy” could be such a softgoal, while “earn more than 2000 €” a hard goal. Plans and goals can be decomposed to subplans and subgoals that can be in AND- or OR-relation among themselves. Resources represent some data or a physical entity, from a database entry to a train ticket. From now if we speak of goals, we intend hard goals.

Different relationships can be modelled: Goals, plans and resources can contribute positively or negatively to the satisfaction of a goal or the satisficement of a softgoal. To understand which plan, goal or resource is the means to satisfy a goal, means-end analysis relationships can be added. Other important relationships model inter-agent dependencies. The actors can delegate a goal or a plan to another actor. Moreover, they can depend one to another for a resource, for the execution of a plan or the achievement of goal (the dependum). Dependencies may be quaternary relationships that include the dependee and the depender actor, the dependum and a fourth argument, which specifies 'why' the actors depend one from another. Again, this 'why'-argument can be again a goal, a plan, or a resource and, in other words, it expresses what entity in the dependee actor needs the dependency. A dependency defines an agreement between two actors, in order to attain the dependum. Goal dependencies represent the delegation of the responsibility to fulfil a goal, needed by the depender, to the dependee. Plan dependencies define activities to be performed by the dependee. Resource dependencies require the dependee to provide a resource to the depender [Penserini et al., 2006c].

2.4.2 Graphical notation

All the concepts and relations defined in the metamodel can be represented graphically in TROPOS diagrams. Tropos diagrams are used in different levels of detail through the different development phases, but with the same notation.

![Graphical representation of the TROPOS concepts.](image-url)
Actors are represented as circles, hard goals as ovals, softgoals as clouds, plans are diamond-shaped and resources are simple rectangles (Figure 2.16). Entities, from goals to resources, can be associated to an actor placing them near it. The entities of an actor can be further detailed in a dashed balloon that is associated to this actor. In this balloon, goals and plans can be decomposed in an AND- or OR-relation, the means to satisfy a goal and contributions (;++:full satisfaction, +:partial satisfaction, -:--: full and partial negative contributions) can be defined (Figure 2.17).

Delegations are represented as an arrow between two actors, with the dependum goal, plan or resource in the middle. The same notation is used for the so-called 'why'-dependencies, only that the starting point for the arrow is not the dependee actor, but the 'why'-argument entity internal to the actor's balloon (Figure 2.18). All dependency arrows have to point directly to an actor. In its balloon, the delegated entity has then to be redrawn and possibly detailed.

Although the same general diagrams could be used for all the development phases, it is used to divide them into actor diagrams that define actors that participate and show
relations between them, and goal- and plan-diagrams that give the main focus on inner-agent goal or plan decomposition and means-end analysis.

A more detailed user guide to TROPOS metamodel and diagrams can be found in [Sannicolò et al., 2001].

### 2.4.3 The development phases

Tropos actually spans over five development phases [Bresciani et al., 2004]: *Early and Late Requirements Analysis, Architectural Design, Detailed Design* and the *Implementation*. In this thesis, the methodology is used with a recent improvement of the Tropos capability definition (see Section 2.4.4). The Detailed Design and Implementation phases are described, by and by, later in this thesis, because design and implementation of the operative part of system functionalities were reengineered along with the introduction of the new capability definition, while the remaining shall be the proposal of this thesis.

#### Early Requirements analysis

This phase models the actual system as it is, without considering the system to develop, to capture not only *what* software needs to be developed, but also *why*. In order to elicit the requirements, the stakeholders are identified and their needs are analysed. In an actor diagram, stakeholders are modelled as social actors that depend on one another for goals to be satisfied, plans to be executed and resources to be provided. For each single actor, the important needs (hard- and softgoals) are detailed in a goal diagram by analysing and decomposing them.

![Figure 2.19: Example diagrams for the first four phases of the TROPOS methodology (from Bertolini et al, 2005).](image)
Late Requirements Analysis
This phase introduces the system-to-be and models it in relation to its environment, extending the actor diagram of the previous phase. The stakeholders’ intentionalities have to drive the design of the system functionalities. To achieve this, stakeholders delegate the goals they want to be achieved, to one or more system actors. These dependencies define the functional and non-functional requirements of the system. Now, in a goal diagram, an internal system analysis is performed, starting from the delegated goals. Goals can be decomposed at a high level and contributions between goals and softgoals are examined.

Architectural Design
In this phase, the system architecture is developed. First, the organizational architecture is defined, decomposing the system into a multi-agent system. New sub-actors are introduced, which are responsible for achieving some goals or performing plans emerged along the system goal analysis made in the previous phase. Additional actors can be needed to contribute to the fulfilment of some specific functional or non-functional requirement or be demanded by a particular architectural style or organizational pattern. The actors introduced in this phase can be seen as software agents or, to develop a more flexible system, as roles, which can be played by an agent. Now goals and plans of new actors have to be further refined on a more detailed level. The functionalities for achieving a goal can be analysed by defining the plans in means-end relation to the goals; then, these plans can be decomposed till down to atomic activities (i.e. other plans).

The next step consists in capturing the capabilities (functionalities) that software agents or roles have to provide and that are further developed in the detailed design phase, using UML v. 2.0 activity- and sequence diagrams. This last step is detailed in the next section.

2.4.4 The new Capability definition
Recently, the TROPOS methodology has been extended with the agent capability modelling technique [Penserini et al., 2006a][Penserini et al., 2006b]. Agent capability has been defined in agent-oriented programming [Shoham, 1993] as the ability of an agent to achieve a goal and refined basing on the philosophical idea that ‘can’ implies both ability and opportunity [Padgham and Lambrix, 2005]. An agent has a “capability for a giv-
en goal”, when it has at least one plan – the ability – that is able to satisfy the goal and user preferences and environmental conditions – the opportunity – are sufficiently satisfied. The design process was revisited in order to make capability modelling and analysis more explicit and systematic.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Means_End (goal,plan)</th>
<th>List of Contributions</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TravelAgent goCrossCountry, goBike</td>
<td>[Fun +1], [minEffort +]</td>
<td></td>
<td>not raining</td>
</tr>
<tr>
<td>TravelAgent carTransport, transportBike</td>
<td>[Fun -], [minEffort +]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TravelAgent goMotorway, travelCity</td>
<td>[Fun -], [minEffort ++]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TravelAgent goStreet, travelMountain</td>
<td>[minEffort -], [Fun ++]</td>
<td>summer</td>
<td></td>
</tr>
<tr>
<td>TravelAgent goStreet, travelCity</td>
<td>[Fun +], [minEffort +]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TravelAgent trainTransport, transportBike</td>
<td>[Fun --], [minEffort ++]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.20: Capability table for a sample agent, generated by Taom4E.

A capability is a couple <goal, plan> that arises from the goal-plan means-ends analysis (Figure 2.21 A) and consists of ability and an opportunity part. For each agent, the identified capabilities are stored, along with their opportunities, in a capability table (Figure 2.20). The ability part is composed by a set of concrete activities that can be modelled by TROPOS plan decomposition diagrams and further detailed following quite standard object-oriented methodologies: From the plan decomposition diagrams, a tool can generate UML 2.0 activity diagrams, where the control-flow structure is modelled (Figure 2.21 B). For each activity in these diagrams, sequence-diagrams can be

Figure 2.21.: Capability: (A) complete view in Tropos notation, (B) activity diagram for control-flow structure of the ability plan, and (C) agent-interaction for one activity of the ability (from [Penserini et al., 2006a]).
generated, in which complex agent-interactions can be put together, combining pre-defined communication protocols (Figure 2.21 C).

An automatic mapping from these UML diagrams to an implementation within a software agent based on the FIPA-compliant platform Jade, is proposed, too, in the abovementioned papers.

The ability part only gives a partial description of the capability, that is, it does not provide any information on how the environment influences the system behaviour of the at run-time. This is described by the opportunity part, which is modelled via plan/softgoal contributions from Tropos diagrams and environmental constraints (e.g. temporal constraints between sub-plans) that are specified by model annotations. In practice, the opportunities can drive the selection of the best ability to achieve a goal, regarding preconditions, positive or negative contributions to other goals or softgoals (e.g. quality of service needs), user preferences, environmental constraints etc. The implementation of the opportunity part has been partially shown in [Penserini et al., 2006b], in this approach the authors illustrate how to map Tropos concepts into Jade structures. Notice that the proposed approach does not deal with any generalized and automatic mapping.

Therefore, this thesis work aim at complementing and improving the previous work proposed in [Penserini et al., 2006b] by allowing the designer to automatically map Tropos diagrams into agent structures, taking into account that opportunities form part of the knowledge level information available at the end of the architectural design phase.

2.4.5 Formal Tropos

To give a formal description to the TROPOS models, the formal language Formal Tropos was defined [Fuxman et al., 2001]. It provides a textual notation for models and allows describing dynamic constraints among different elements of a specification in a first order, linear-time temporal logic. Formal Tropos has precisely defined semantics, amenable to formal analysis techniques, in particular model checking. A tool is provided, that supports the specification of a model using this formal language and extends the symbolic model checker NuSMV, for consistency checking, property validation, and simulation of the specifications.
The visual modelling tool Taom4E [Perini and Susi, 2004][Bertolini et al, 2005] was developed at ITC-Irst to support the TROPOS modelling activities. It is a plug-in for the Eclipse platform. Taom builds on the Eclipse plug-ins EMF and GEF and implements the whole Tropos metamodel. Tropos actor diagrams can be graphically created and extended for early and late requirements analysis phases. Each actor can be detailed in a goal diagram, shown in a “balloon”, where delegated goals are displayed and can be decomposed. System actors can then be detailed to a multi-agent system, in architectural design diagrams. For every agent the capability table can be shown, models can be exported as jpeg image and stored as OMG-standard XMI files with extension *.tropos. Figure 2.1 shows a sample Taom diagram window, using the associated perspective. The version used for this project is Taom4e 0.4.0, with support for capability detailed design and implementation.

1 “Tool for Agent Oriented visual Modelling for the Eclipse platform”, (http://sra.itc.it/tools/taom4e).
2 Eclipse is an plug-in based, multi-platform, open-source development environment. See http://www.eclipse.org/ for details and download.
3 Methodological Approach

The proposed approach is collocated at the end of the architectural design phase in the TROPOS methodology and aims at design and implementation of the software agents in the system-to-be. In particular, we focus on architecture and dependencies of a single actor, described in a goal diagram.

In a goal diagram, delegated goals are decomposed and analysed, and the functionalities are modelled, the actor has to provide to satisfy that goals. The TROPOS agent capability modelling technique (Section 2.4.4) gives a straightforward approach to capture the functionalities delivered by an actor, and on the same time a simple proceeding to separate these abilities from the actor's knowledge. Therefore, at the end of the phase, the diagram can be seen as split into two parts (Figure 3.1):

One part is the ability part of the capabilities. The area of an ability begins, by construction, at a plan in means-end relation to a goal (this signifies that the plan is capable of satisfying that goal), and includes the plan decomposition hierarchy, that is transformed into an activity diagram. The single abilities can be treated and eventually implemented, following the process proposed in [Penserini et al., 2006a][Penserini et al., 2006b].

The other part is all the remaining model: from the leaf goals connected to the abilities up to the agents with their dependencies. This information, modelled by goal-diagrams
and inter-model dependencies (why links), are strictly related to the domain knowledge that such an agent owns in terms of main objectives and responsibilities (goals), activities to fulfil in order to achieve such objectives (plans), social relationships and commitments (‘why’ dependencies), etc. In the following chapters, we will see why it is important to map to the runtime instance of an agent not only the functionalities, but also the information given by the knowledge level and how this will be achieved.

The separation of the knowledge part from the operative part for each agent capability has some additional benefits: It facilitates reuse, testing, and maintenance of agent capabilities. On the other side, following this technique, the use of some relationships is restricted, because connection between ability- and knowledge-level is almost limited to means-end-relations.

**Agents and Roles**

Before proceeding, one more consideration has to be made. As already said, in TROPOS an actor can represent an agent, a role or a position. In the following chapters, we call all actors modelled in the system agents, disregarding the fact that they could represent roles or positions. For the purpose of this work, a role represented by an actor can also be conceived as an agent that plays only this single role. Later on, the use of roles will be further discussed.

**3.1 The knowledge level in TROPOS**

The first step of the approach is to figure out what knowledge level information a TROPOS model provides and why it is important to use this information in the system implementation.

- Analysing the goal diagram model bottom up, we have first abovementioned capabilities with their opportunity part. They contain knowledge about which abilities can be executed to reach a goal (from the means-end analysis) and the conditions, which may drive the run-time selection of the best ability: softgoal contributions and annotations to express environmental constraints, user preferences and the like. The plan selection could be guided, giving a precondition that enables or disables the achievement of specific softgoals.

- Next, we have the goal hierarchies – root goals, AND- and OR-decomposed into more specialized goals, down to the leaf goals that are part of the capabilities. Until
now, these hierarchies were used only for architectural design purposes, to catch the functionalities needed to achieve the root goals. Bringing the whole goal hierarchies to the run-time instances of the agents, the additional knowledge on the goal decomposition process could be useful to react better to environment changes and to possible changes in the available functionalities and in the goal hierarchy.

• Agents can depend upon another for the satisfaction of some goal, for some plan or resource. Dependencies on the goal level and delegations define high-level interactions between agents. They can be seen as stakeholders' requests for goal achievement. Besides, 'why'-dependencies can deliver additional information on the goal decomposition process.

• Contribution links between different goals and softgoals are another important concept at the agent knowledge level. If the importance of goals or softgoals changes during the agent life cycle and the run-time agents are aware of this knowledge, the execution of functionalities can be dynamically adapted to better satisfy the important goals. [Penserini et al., 2006b] states that a softgoal is suitable to model a system strategy in order to deal with an adaptive behaviour, driving the plan selection according to the contribution link analysis. This can also be extended to the goal selection.

3.2 The knowledge level at run-time

The last chapter showed, which information, contained in a TROPOS diagram, belongs to the knowledge level. In this chapter we analyse, which basic reasoning abilities an agent would need to use its abilities at best, and understand, why it is important to bring knowledge from modelling into run-time instances.

From the viewpoint of the implementation, the automatic code generation process presented in [Penserini et al., 2006b] delivers all the functionalities (the abilities) of a system actor as behaviours that can be used by an (Jade) agent. However, from the point of view of this agent now many questions arise:

What are the goals to achieve? When and in which order have the functionalities to be executed? How can a selection between two or more functionalities be made, if they can achieve the same goal? How to react to failures and environmental changes? To cope with these issues, we focus on the
agent's reasoning part that has to make all these decisions to optimally satisfy stakeholders' needs.

We analyse, to what extent the knowledge needed to take these decisions can be extracted from the developed TROPOS models and how the agent has to be engineered to use this knowledge at best. For a first approach, we suppose that all goals are achieve-goals. This signifies that the objective of the agent is to achieve the goal whenever it is requested by the stakeholder, and does not try actively to maintain this state (in contrast to maintain-goals).

The goals that have actually to be achieved can be communicated by the user with a request-message. Hard-coding the type and sequence of functionalities to be executed (maybe extracted from the design model using some planning algorithm) in order to achieve that goal, you loose some of the knowledge contained in the model. This thesis proposes an agent architecture that contains the opportunity part of the capabilities and the whole goal hierarchy with all modelled dependencies and relations. Being aware of all the knowledge modelled in the design phase, the agent is flexible (having multiple ways of achieving a goal) and able to react to failures and to changes in the environment exactly in a way that meets the designed requirements, two important properties for an agent system [Padgham and Winikoff, 2005].

Moreover, being aware of the whole goal structure, the agent is prepared to make dynamic changes of its own structure, adding functionalities or dependencies or refining goal decompositions, to have a certain learning ability.

Using the same representation of knowledge from the analysis to the implementation gives the possibility to track decisions made on run-time up to the requirements engineering phases and vice versa. So, in an iterative development process, changes on the design can be directly mapped to the implementation, while agents' run-time decisions can be considered to refine the architectural design model.

3.2.1 What we are trying to achieve – and what not

In our approach, in contrast to other proposals, e.g. [Cheong and Winikoff, 2005], we try to avoid additional diagrams or non-standard annotations at the design phase. This should be possible by using the capability modelling technique, which naturally forces the conceptual division of goals and plans in the design process and strengthens the division of the procedural ability-parts (just modelled) from the reasoning parts. Like
already said, this work does not deal with the abilities part that is modelled using additional UML 2.0 diagrams, in the detailed design phase, and already mapped to code as detailed in [Penserini et al., 2006a]. Section 4.3.4 shows the integration of this part with the implementation proposed here.

We want to give to TROPOS structures semantics that follow the semantics attributed to them in practical use and do not want to adjust TROPOS to our needs to achieve an optimal implementation. Moreover, only few annotations are added to TROPOS models and no additional diagrams were used.

Furthermore, we are interested to build up an agent framework that is flexible enough to support the general properties discussed above, e.g. despite of domain specific AI techniques (DCBR, Planning, Machine Learning) are outside our scope, the proposed agent framework could possibly embed them. We focus on the automatic mapping process, implementing, where necessary, some simple decision-making algorithms that use the knowledge available at architectural design. Therefore, the scope of this work shall not be to develop algorithms to find the globally best way to satisfy all possible goals, as already investigated and implemented in the Tropos group by the goal reasoning tool “GR-Tool” [Giorgini et al., 2004].

Finally, in this preliminary effort, we do not aim to map the whole TROPOS model along with all its design possibilities and its structure expressiveness to the implementation. We need to have a small set of modelling concepts to work with, to understand the basics of and define a mapping to an agent-oriented implementation. The next section shows, which components and relationships are important in a model at the end of the architectural design phase. They should be appropriate to characterize a multi-agent framework and the agent’s reasoning part, with focus on BDI-like agent-oriented concepts.

**3.3 Preparing the TROPOS concepts**

Following the TROPOS metamodel, a multitude of different relationships between all entities (actors, goals, plans and resources) can be modelled. In order to extract informations from the TROPOS goal analysis to build a rational BDI-Agent system, precise semantics for a basic set of TROPOS knowledge level concepts have to be defined, avoiding semantic overload and ambivalence. Besides, we focus on concepts, which represent at most the agent-oriented paradigm.
An attempt to limit the relationships usable in Tropos models was proposed by Hugo Estrada [Estrada, 2006b]. This approach goes top-down, removing from the metamodel all relations that are redundant or that have no meaningful semantics. For this thesis, a pragmatic bottom-up approach is used, trying to build a small, but not necessarily minimal set of well-defined basic concepts, that is still usable and intuitive.

### 3.3.1 The diminished set of TROPOS relationships

The proposed reduced set of Tropos modeling concepts considers these properties: well-defined semantics, a simple and clear design, adequate expressiveness and an easier implementation of the prototype. Focusing an automatic implementation, the set is mainly restricted to a goal diagram, the knowledge level of a single software agent (a system actor).

This reduction aims at a model at architectural design, which should be the base for the construction of a software agent. We do not want to forbid the use of the other concepts in earlier design phases and during the design refactoring or refinement.

The diminished model contains the following:

- The basic Tropos constructs actor, goal, softgoal, plan and resource.
- A set of relationships with limited possibilities of use between these constructs:

  1. **AND/OR-Decomposition**: A homogeneous\(^1\), acyclic relation to decompose goals, softgoals, plans and resources hierarchically into sub-entities.

  2. **Means-End-relation**: A relation between a plan (the *means*) and a goal (the *end*). The other combinations that are possible in the original metamodel, are not allowed. Relations between softgoals and goals, with the same semantics, can be modelled more accurate with contributions, while homogeneous relations can be modelled using OR-decompositions.

  3. **Use- and Produce-relation**: A special means-end relations between a resource and another entity. If the resource is the *means*, the relation is labelled as use-, if it is the *end*, as produce-relation (first proposed in [Estrada, 2006a]).

  4. **Delegation**: A relation between two actors, to fully delegate goal, softgoal, resource or plan to another actor. Focusing on interesting agent-oriented paradigms, for the proposed prototype the dependum is restricted to a goal.

\(^1\) homogeneous: a relation restricted to entities of the same type, e.g. goal – goal.
5. **Dependency**: A relation between two actors, with a goal, resource or plan as dependum and a goal or plan as ‘why’-argument. Focusing only on the knowledge-level and on the AO-paradigm, for the prototype both arguments are restricted to a goal.

6. **Contribution**: In this first definition contribution relations can go only from a plan or goal to a softgoal and have the values “+++”, “+”, “-”, and “--”.

All relationships are non-reflexive (an entity cannot be related to itself) and can be $n:n$ (e.g. a goal can be composed to $n$ subgoals and, vice versa, subgoals can belong to more than one decomposition, having $n$ parent goals).

**Combining relationships**

In this diminished set, decomposing a goal or plan, no combination of different types of relationships (AND/OR/Means-end) is allowed. For example, if a goal is AND-decomposed, it cannot have a plan in means-end. The only exception is the 'why'-dependency, which can be added to any decomposition. Dependencies have anyway to be fulfilled to satisfy the dependent goal or plan. If other combinations are needed, they could be modelled inserting dummy goals or plans (e.g. Figure 3.2).

![Figure 3.2: Modelling relationship combinations in the diminished Tropos model.](image)

### 3.3.2 Semantics, approaching an implementation

Now, focusing one agent, for each relation precise semantics are defined, avoiding semantic overload and ambivalence and, where necessary, justifying the restrictions made.

**AND-Decomposition**

This is a complete\(^1\) decomposition of a goal into subgoals or of a plan into sub-plans. **All** sub-entities have to return successfully to achieve the root goal or plan. Possibly the

---

\(^1\) complete: The whole activity is delegated to the subgoals/plans. In practice the root has the selection, execution and monitoring control.
Methodological Approach

execution order can be explicit. The same applies to the decomposition of softgoals and resources, but these concepts are irrelevant for this prototype implementation. Indeed, inhomogeneous decompositions are not considered, because they can be modelled more precisely using means-end relations or “full” contributions (with value “++”).

**AND-decomposition**

Like the AND-decomposition, a complete decomposition of a goal into subgoals or of a plan into sub-plans. If at least one of the sub-entities returns successfully, the root entity is achieved. In theory, the selection of which sub-entity to activate, is non-deterministic. All entities get activated and with the first one that returns, the root is achieved. However, in practice we need some algorithm to deal with selection and error handling. With some metrics to classify sub-entities, the selection algorithm can select the best sub-entity and, on a failure, the best one remaining.

![Figure 3.3: Allowed AND-decompositions.](image)

**OR-Decomposition**

Like the AND-decomposition, a complete decomposition of a goal into subgoals or of a plan into sub-plans. If at least one of the sub-entities returns successfully, the root entity is achieved. In theory, the selection of which sub-entity to activate, is non-deterministic. All entities get activated and with the first one that returns, the root is achieved. However, in practice we need some algorithm to deal with selection and error handling. With some metrics to classify sub-entities, the selection algorithm can select the best sub-entity and, on a failure, the best one remaining.

![Figure 3.4: Allowed OR-decompositions.](image)

**Means-End**

This is a relation between a plan and a goal, where the plan is the means to achieve the goal. A goal can have more means that achieve it and plans can be means for the achievement of different goals. This achievement is always total, i.e. if more plans are in relation with a goal, these plans are in an OR-relation against each other. To select the plan to execute, the same criteria, as for OR-decomposition can be used. The means-end relation is part of the capability definition, giving the workflow of the activities that
3.3 Preparing the TROPOS concepts

can be executed to achieve a goal. The relation is restricted to goals and plans. If similar semantics are requested to model relations between goals and softgoals or between plans, the use of decomposition- or contribution links should be considered.

Use and Produce

As relation from resources to goals and plans the means-end link gets the semantics of “uses the resource” if the resource is the means, and “produces the resource” if it is the end. This relation would be useful for an implementation, to model data passed between plans or environmental concepts that can be “used” for goal- or plan-selection. Unfortunately, the “produce”-relation is yet not allowed by the actual TROPOS metamodel.

Contribution

The contribution relation is considered in the prototype only if drawn between a goal or plan and a softgoal. By its nature, softgoals cannot be completely satisfied, so contributions are semantically correct relations to indicate the degree of their satisficement. The contribution relation is often used to evaluate non-functional (quality) requirements [Bresciani et al., 2004][Penserini et al., 2006b]. Contributions between plans and softgoals are part of the opportunity of a capability, and, as proposed in [Penserini et al., 2006b], one of the metrics for deciding, which of the abilities available to achieve a goal, is the most opportune to execute. Contribution
links from goals to softgoals were added with the same aim, i.e. mainly to give decision-making support for OR-decomposed goals.

Delegation

Delegation is one of the key principles in the earlier phases of the methodology. At the end of architectural design, focusing on a single agent, the most important delegation is the delegation of a goal from a stakeholder or another system actor. The delegated goals are analysed and decomposed during architectural design phase and have to be satisfied at run-time by the dependee actor. If the goals are annotated to be achieve-goals, the software agent has to try to satisfy them every time it is requested by the delegating actor (in small systems usually a stakeholder), if they are maintain-goals, the agent has to act proactively to maintain the state given by the goal satisfaction conditions. From the point of view of the delegating actor, the delegation is “full”, i.e. this actor has no more responsibilities regarding the goal satisfaction.

'why'-Dependency

The 'why'-dependency (see Figure 4.6, p. 71) is an important concept to model dependencies between actors in the system. The following are templates for a dependency between two actors A and B with goal g or plan p as 'why'-argument. The 'why'-argument delivers the reason for being dependent – “A depends on something in B, because g or p needs it to be accomplished”.

- A depends on the fulfilment of a goal by B because g needs it to be satisfied.
- A depends on the execution of a plan by B because g/p needs it to be satisfied/to be carried out successfully.
3.3 Preparing the TROPOS concepts

- $A$ depends on the production of a resource by $B$ because $p$ needs it to be carried out successfully.

An important consideration has to be made regarding how the subgoals or -plans of $g$ and $p$ are related to the dependency. From the use of the expression "need" in the sentences above we can conclude that, for $g$ and $p$, each dependency is in an AND-relation with the other dependencies and with the set of subgoals and -plans. From the viewpoint of the dependee actor, the 'why'-dependency is handled at the same way as a delegation: the dependum has to be analysed in the architectural design phase and at run-time the agent has to try to satisfy it every time it is requested.
4 Mapping and Implementation

In this Chapter, using the semantics previously defined, a mapping from the (restricted) TROPOS model to an agent-oriented platform is proposed. More formally, we go from a platform independent model (PIM) to a platform specific model (PSM). Specifically, the proposed approach considers Tropos goal diagrams from the architectural design phase as the PIM model. The agent implementation we want to obtain with this work naturally accommodates with a BDI-architecture (Section 2.1.1, page 17), that provides the concepts of goal, plan and belief. The BDI-oriented platform Jadex (Section 2.3.2) was selected due to its goal handling concept, and because it is open-source and built on top of the widely used, FIPA-compliant agent platform Jade. Jadex differs from other BDI frameworks by its notion of goals that are not only transient events, but an agent is aware of its goals at any time. This facilitates the mapping of basic modelling concepts in Tropos goal diagrams directly to the implementation.

The focus is given on the implementation of one agent of the MAS, as agents are autonomous and there is no global control, so every agent can be implemented independently. Specifications and requirements for the implemented system and for a code-generation process are given. Then, a mapping from the knowledge level of Tropos to the platform-specific is given, followed by the definition of a concrete code generation process. A prototype for the mapping phase has been developed to automate this process. Finally, the generated code is integrated with the abilities part of the capabilities.

4.1 Specifications and Requirements

What does the developer demand from an automatic implementation? Basically, the implemented multi-agent system has to deliver the correct output and to be flexible enough to cope with user needs (by request messages) or environment properties changes. The agent has to be aware of the Tropos goal analysis results (i.e. design time properties) in order to select the right abilities; namely, it has to select the subgoals to pursue, according to the semantics of the goal decomposition relations and related opportunities till down the selection of different capabilities, monitoring failures and managing failure recovery. All this has to be carried out, taking account of user prefer-
ences and quality of service needs, in an environment that can change at runtime, even during the goal achievement process.

The automatic code generation tool should help the programmer with the creation of the basic agent structure. It should help to deliver a Jadex-based program, strongly reducing the coding effort. It should be a road towards the solution, even for not-very expert agent programmers, without trying to build the final solution asking lots of parameters or additional diagrams, to build deliverable code.

Following, some requirements worked out for the different steps of development of the system.

**Requirements on the mapping**

A BDI-based Jadex agent has to be built using all information that is retrievable from a Tropos goal diagram (restricted to the previously defined diminished model), focusing only on the knowledge level, not the plan decompositions (ability part).

Goals, plans, resources, relationships between these concepts, and dependencies between agents have to be represented, maintaining their semantics. The mapping has to use and respect as far as possible the concepts provided by the Jadex platform and to extend them where necessary. Indispensable details that affect the agent's overall structure could be annotated in the model.

**Requirements on the generated code**

The generated code should use as much as possible the possibilities provided by Jadex, i.e. the concept of meta-reasoning, triggering, the functionalities provided by the belief base, etc. The mapped information should be gathered, as far as possible, in an ADF file. This allows our approach to be as much as possible independent from any BDI implementation, as specified in the MDA approach.

The code should deliver a clear and flexible interface where the programmer can put business logic, a clear interface where to extend/adapt the goal and plan selection algorithms and criteria to particular needs. This is also important in order to support agent knowledge sharing, e.g. learning new capabilities. Furthermore, the generated code should not restrict the programmer in adapting or changing the generated system, in adding new concepts or modifying them using the facilities available on the Jadex platform.
**4.1 Specifications and Requirements**

**Requirements on the code generation tool**

The code generation tool should take in input Tropos diagram XML files generated by Taom4e and generate all files necessary to build an agent structure executable “out of the box”. The generation process should be straightforward, without demanding user interaction to provide additional information. Preliminary testing facilities have to be provided, to verify that everything behave as designed. Eventually, tool has to produce code to enable an automatic integration of capabilities generated with the process described in [Penserini et al., 2006b] and has to be integrated, as an eclipse plug-in, into the TAOM4E tool.

**4.2 The mapping**

This section gives a transformation from the previously defined diminished Tropos model to BDI-based structures for a platform specific implementation on the Jadex platform.

Jadex allows a direct mapping of goals and plans, but is not able to deal with some of the Tropos concepts, such as the hierarchical goal structure. Briefly, decompositions and means-end relations are mapped using the Jadex triggering mechanism, some additional “service”-plans, and some control code, added extending Jadex Plan classes. The belief base plays an important role for storing resources, relationships and opportunities. Besides, dependencies become simple requests to other agents.

**4.2.1 Characteristics of Jadex, focusing on the mapping**

Before proceeding with the mapping, some important characteristics of the Jadex architecture are examined.

Jadex distinguishes goals, plans, and (external) messages, whereby every one of them has different activation mechanisms and properties. To map a goal hierarchy directly to Jadex, Jadex goals would have to be able to accept external request messages, to activate their subgoals, and to execute control- and failure-handling code. However, in Jadex, these key activities can be pursued only using plans.

For plans, a goal or an external message can be set as trigger. This means that every time, the associated goal- or message-event is fired by the platform, Jadex executes the
Java code that implements the plan, that is, the `body()`-method in the Java class associated to the plan through the ADF.

![Diagram](image)

**Figure 4.1:** Possibilities for goal dispatching and plan triggering in Jadex

Goals are usually *dispatched* by the Java-code within a plan. In alternative it is possible to dispatch them on the initial state or by giving a creation condition that monitors belief base values. The creation condition seems promising to dispatch goals from other goals, without a plan. Nevertheless, the problem of this approach is that goals cannot modify belief base values. Thus, you cannot avoid using plans to dispatch goals.

Another conceptual problem concerns the ability to pursue goal hierarchies at runtime. To “connect” plans and the goals that trigger them, Jadex uses the aforementioned “bottom-up” triggering mechanism and you have no possibility, having a goal, to know at runtime, which plans have set this goal as their trigger. The developers of the Jadex system replied to a request, that this is a conceptual problem, as a goal should not know in advance which plans can satisfy it. As a workaround for this problem, we had to decide between two design choices: hard-code links between goals and plans in Java or map the hierarchies to belief base entries in the ADF. The second solution was preferred, because she is more flexible and respects the requirement to gather as much as possible mapped information in the ADF file.

### 4.2.2 Entities mapping

**Plan**

This mapping considers only the plans that have a direct means-end relationship to some goals (that correspond to the root plans of a capability). In practice, these plans can be seen as the attachment point for the abilities mapped to the implementation, as
detailed in [Penserini et al., 2006a]. **Plans are mapped to Jadex plans**; their associated code can be completed with business logic (conceptually, the abilities). Several other types of Jadex plans will be introduced in this section, to distinguish them, these plans are called *real plans*.

**Goal**

**Goals are mapped directly to Jadex goals.** As discussed above, a result of the Jadex goal characteristics is that goals need to be supported by plans in order to enhance their abilities, to map correctly all relationships. Additionally, goals are mapped to the belief base, used to build the goal hierarchy. It is important to observe that a goal fails if no one of the plans it triggers succeeds.

**Goal types**

Although the mapped code is flexible enough to handle all goal types provided by Jadex, in the current automatic mapping we have supposed that all (Tropos) goals are *achieve-goals*. The only use of achieve-goals gives to the implemented agent a solely reactive behaviour. The agent waits for goal achievement requests and tries to satisfy them. On the other hand, *maintain-goals* are essential concepts to build proactive agents that pursue goals without requests from outside. Maintain-goals pursue activities to maintain a well-defined condition in the environment. As soon as this condition is achieved, they are reactivated every time the interested part of the environment changes.

The goal decomposition diagrams are analysed to understand, when goals can be classified as *maintain-* or *achieve-goals*, figuring out some mapping possibilities. Unfortunately, they could not be applied without giving restrictive limits to the software engineering process. The remaining solution — annotate the Tropos goal diagram — was not tested, although the mapped code, the generation tool and the Jadex platform were prepared for the use of maintain-goals. However, the maintain condition to maintain has to be added by the developer at design time.

**Resource**

Resources map naturally to an entry or a set of entries in the belief base. Since in Jadex the belief base is an object-oriented database, the entry can be related to an arbitrary Java object. Resources should be changeable at runtime with a user request message, to reflect changes in the environment.
Softgoal

From the implementation viewpoint, softgoals are abstract entities. They are not goals to achieve, but mainly used to define opportunities for the selection of the next goals or plans to pursue, if a goal is OR-decomposed or if there is more than one plan in means-end relation to a goal. A softgoal is therefore mapped only to a belief base entry, which contains its name and a value that may be changed by the user at run-time, namely such a value expresses the softgoal actual importance. It would be desirable to put this information as a property of softgoals into Tropos diagrams.

4.2.3 Relationships mapping

Means-end relation

The means-end relation can be mapped—one to one—to the plan triggering mechanism, one of the key features of Jadex. Having defined no conditions, every time the associated goal is activated, plan execution is triggered (all goals are applicable). Corresponding to Tropos means-end analysis, Jadex supposes that every applicable plan for a goal (without achievement conditions) is able to satisfy that goal completely. Therefore, if more than one plan is applicable, meta-reasoning (explained in Section 2.3.2) is utilized.

![Diagram](image)

Conforming to the semantics, plans should be selected according to highest contributions. Therefore the default meta-reasoning selection algorithm, provided by Jadex, has to be replaced by adding for every goal a meta-goal and its meta-plan (Figure 4.2). This first mapping proposes a very simple calculation of measurable values, multiplying the contribution (value) with the softgoal importance. The plan with highest value has to be
selected for execution. Environmental conditions, defined in some capabilities, could be mapped to plan creation conditions containing references to belief base entries that represent environmental facts.

**Failure handling:** Failures are handled taking full advantage of Jadex mechanisms. If the execution of a plan fails, meta-reasoning is automatically reactivated. The meta-plan can select only between applicable plans, and failed plans are (by default) no more applicable. The meta-reasoning concept has only one drawback: if only a single plan is applicable, this plan is automatically executed without executing the meta-reasoning plan. Thus, this plan is not suited to be monitored and to log failures. In the real plans, logging has to be made, before and/or after the business logic was performed. If all plans fail (therefore no more plans are applicable), the goal returns a failure. This mechanism is entirely handled by Jadex.

**OR-decomposition**

As previously seen, in Jadex goals cannot “activate” other goals, but only be the triggering event for a plan. So, to map the goal decompositions, some “service”-plans have to be linked between goals. OR-decompositions are mapped, inserting a plan between each relationship. As illustrated in the right hand-side of Figure 4.3, this *dispatch-goal plan* (green hexagon) is triggered on the activation of the parent goal and it dispatches one subgoal. Now parent-goal and *dispatch-goal plans* form the same structure as a means-end relationship and can therefore use the same logic, for plan selection and for failure handling.

One consideration has to be made about the selection criteria: What are the opportunities for goal selection? Using the same simple method as with means-end relationships, opportunities could be calculated basing on softgoal contributions of subgoals. We decided to use a more general decision algorithm that takes into account the whole hierarchy under the parent goal. The contributions at all levels, till down to plans in means-end, are summed up. For OR- and means-end-decompositions the highest contributing value is selected, for AND-decompositions the mean value. Only 'why'-dependencies cannot be considered, because this would comprise requests to other agents, possibly requiring high computational and network load, and taking into account possibly incompatible softgoals from these agents. That is, the design establishes that the dependum of a ‘why’ dependency is considered delegated to another agent, therefore it is not the responsibility of the current agent any more, even if it is necessary to achieve
Mapping and Implementation

an internal goal. The algorithm takes advantage of the consideration that a contribution to a softgoal at higher level has the same weight as the one obtained by the contributions calculated from all entities on a sub-level to the same softgoal. Therefore, contributions at a higher level can be pushed down to the leafs without changing the semantics of the modelled system (see Figure 4.4 for an example). It can still be used for means-end reasoning, without changes.

**Figure 4.3:** The mapping of the OR-decomposition relation from TROPOS to Jadex.

To make a navigation of goal hierarchies possible, according to the considerations made at the beginning of this chapter, all affected relationships have to be additionally stored in the belief base. For the goal and plan selection, another possibility would be to deal with costs for the execution of every plan. Costs include softgoal contribution and importance; negative contributions could cause higher cost. They can be requested to other agents, summed, and transparently modified by the programmer.

**Figure 4.4:** Considered equivalence of contributions to different levels in the goal hierarchy.
**AND-decomposition**

If an AND-decomposed goal is activated, all subgoals have to be dispatched. This cannot be realized by triggered plans – not by using meta-reasoning, neither by setting the property that all triggered plans are executed – it does not allow failure monitoring. As illustrated in Figure 4.5, the following solution was adopted: An AND-decomposed goal is set as trigger for exactly one plan, called *AND-dispatch-plan* (green hexagon). In the plan body all subgoals have to be dispatched in (some, perhaps random) sequence, if one subgoal fails, the process has to be stopped and a failure has to be returned. For this first proposal, on failure no attempts for rollback of already executed actions have been considered.

![Figure 4.5: The mapping of the AND-decomposition relation from TROPOS to Jadex.](image)

The mapping supports priorities. If they are set as Tropos annotations or by the developer during the mapping, they predefine the goal dispatch sequence.

**Contribution**

The use of contributions was already discussed as selection criteria for OR- and means-end relationships. Contribution links are mapped to an object in the belief base, containing starting and ending point (a goal or plan and a softgoal) and the contribution weight (“++”, “+”, “-”, or “--”).

**Delegation**

For the purpose of the prototype, and having only achieve-goals, we see pure delegation as a relationship between the user and the software agent, where the user makes re-
quests to the system to satisfy a goal. Therefore, the system has to be able to receive goal satisfaction requests. To each delegated goal a plan, called *request-plan*, is associated, that can be triggered by a request message for that specific goal. This plan has to handle the standard FIPA-Request interaction protocol, informing the user for acceptance or rejection of the request, and it has to dispatch that goal and to return success or failure information regarding goal achievement.

Multiple delegation – the delegation of an user-requested goal from one software agent to another, could be easily mapped in a similar manner, but this type of delegation is actually not allowed with the Tropos metamodel in use for the Tropos visual modelling tool TAOM4E.

**'why'-Dependency**

Dependencies between two system actors with 'why'-argument can be easily mapped to a FIPA-Request interaction, like a delegation. The dependent actor has to make a request to the dependee. This request could contain a goal, a plan or a resource, but for the prototype, the mapping is limited to goals, as asking for goal satisfaction is the most agent-oriented concept, whereas the other request could be also seen as “call” for a method or some data.
4.2 The mapping

A special service plan has to be associated to the goal that needs the dependency. This plan has to carry out the side of the initiator for the request protocol. Its counterpart, i.e. the dependee, can be realized implementing the same request-plan seen for the delegation. If the dependum goal cannot be achieved, the dependent goal has to return a failure.

**Use/Produce relationship**

The use- and produce-relationships between plans were mapped and implemented manually, for testing purposes, although actually they are not supported by the Tropos metamodel. The relationships map directly to the code in the *real plans*. They consist of simple convenience methods that read and write resources respectively from the belief base. Interesting information for the mapping process could be extracted from combinations of use- and produce-relations between plans or goals in an AND-decomposed hierarchy. Specifically, they could define goal dispatch or plan execution orders without introducing specific annotations.

---

*Figure 4.6.: Example mapping of a "why"-dependency relation from TROPOS (drawn with Taom4E) to Jadex.*
4.2.4 User interaction

The mapped agents do not act active within the environment and have therefore to be able to react to requests for goal satisfaction from outside. In our implementation, these requests have to be FIPA-compliant “Request”-messages containing the goal name and some parameters. Nevertheless, to affect selection between goals or plans, it would be convenient to define a more expressive requests format, to express preferences for goals and plans, or to give preconditions. This can be expressed in an indirect way, changing the importance of softgoals. Besides, preferences and preconditions could be communicated to the agent as environmental changes that map to the belief base. Defining for goals or plans creation conditions, which refer to these values, on OR- and means-end decompositions plan- and goal-selection can be driven. More complex requests that involve possibly more goals, would need specialization of the goal dispatching logic by the developer.

4.2.5 Agent roles and Jadex capabilities

Until now, all Tropos actors were supposed to define agent classes that are instanced on the Jadex platform. If we decide that our Tropos actors define agent roles, this can be modelled using a Jadex feature, named “Capability” (not to be confounded with Tropos capabilities!). On mapping roles to Jadex capabilities, it is possible to assign different roles to one agent and also to add or drop roles at run-time. Jadex capabilities are defined, like agents, in a ADF file containing beliefs, plans, goals, events and expressions and could be generated using the same mapping. An agent ADF file (that does not contain the sections just mentioned) contains links to all capabilities, the run-time agent should have. This mapping was tested on the implementation of the example presented in section 4.2.6.
4.2.6 A mapping example

This section shows an example for the mapping of a goal decomposition diagram of a single agent from Tropos (Figure 4.7) to Jadex concepts, graphically represented in Figure 4.8. The example models an agent which offers an enhanced encyclopaedia search service. It provides three general search types: a general search, search on different encyclopaedia or a search on Internet websites. We suppose, that these goals have been delegated from a stakeholder. Encyclopaedic search is divided into search on Encyclopaedia Britannica and search on Wikipedia. Internet search consists of a search on search machine and parsing of the results found on the favourite website found. These two goals are in an AND-relation and the resource dependency gives us the right execution order: first, interesting web-pages are searched in a search machine, then they are passed to a (imaginary) parser, that extracts relevant results.

Figure 4.7.: An example: TROPOS goal diagram of a simple search agent.
Users can send general search requests or define a preference for search on Internet or pure encyclopaedic search. Contributions arise from some goals and plans toward two softgoals “minimal cost” and “reliable results”. The agent, implemented with minimal additional business logic for testing outputs of different search types, runs as required and compliant with previously defined semantics and failure handling.

This simple graphical example was used to validate and refine the mappings and to develop a first, manual implementation. Its implementation was source for the basic ideas and templates transferred to the automatic code generator.

4.3 Implementation

In the previous section, TROPOS concepts were mapped to Jadex concepts. Next, a concrete mapping to code (Java and XML) is proposed, developed aiming at an automatic code generation process. A working code generator, that takes TROPOS diagrams (modelled with Taom4E) in input and outputs the now defined Jadex code, is presented afterwards.
4.3 Implementation

4.3.1 Agent Definition File (ADF) structure

The Agent Definition File is an XML file that contains the whole agent architecture and is the starting point for implementation and execution of a Jadex agent. A short overview to the overall structure of the ADF file used for the following implementation, is given. The ADF file is divided into eight sections:

- **<imports>** a list of imports, like Java import, to recognize used Java classes
- **<beliefs>** facts (objects) contained in the belief base
- **<goals>** the agent's goals
- **<plans>** the agent's plans
- **<events>** recognized FIPA-messages
- **<expressions>** predefined OQL-like queries to the belief base
- **<properties>** general execution properties
- **<initialstates>** the agent's initial state, empty in the proposed implementation

Imports, expressions, properties, and initialstates contain default values that are independent from the mapped diagram. The expressions are explained as soon as they are used in the plan code. The belief base contains a fixed set of beliefsets, consisting of facts that contain the mapped beliefs. The goals section contains mapped Tropos goals and all needed metagoals, whereas the plans section contains all plans defined by the mapping, grouped by types (real plans, request plans, meta-plans,...). Finally, the events section consists of a fixed section containing standard FIPA-message templates (request, inform, agree, failure, not-understood and refuse) and a generated section containing goal request messages. Jadex goals, plans and events follow a systematic naming schema that uses Tropos goal/plan names and plan types.

4.3.2 Coding the mapped Jadex agents

The conceptual mapping proposed in Section 4.2 is now concretized, writing Jadex code with the semantics defined by the Tropos model. To comply with the requirements and to simplify code generation process, reviewing and modification of the generated structures, the whole mapped code is gathered into an ADF file. This is possible due to the use of specialized Java Plan files for every one of the previously introduced plan types. For all mapped concepts, proposed XML code and connected extended Plan classes are described. Example code is shown and some important archi-
tectural decisions are observed in detail. The implementation was carried out on the Eclipse development platform, the code written is compliant to Sun Java™ version 1.5¹ and uses the new features of this version. Jadex is used in version 0.941, compatible to Jade 3.4.

**Plans**

A Tropos plan is mapped to a Jadex plan, named realPlan_{plan-name}. The `<trigger>` tag contains references to the goals in means-end relation. For the prototype, a simple String is defined as input parameter. It is connected to the goal(s) in means-end relation with this agent. Result can be returned using the `result` parameter. The plan executes the `body()` method of a Java class named like the plan itself that extends `RealPlan` and is contained in the same package as the ADF (otherwise, the package has to be added to the `<includes>` section).

```xml
<plan name="realPlan_Planxy">
  <parameter name="param" class="String">
    <goalmapping ref="MyGoal_1.param"/>
    <goalmapping ref="MyGoal_2.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="MyGoal_1.result"/>
    <goalmapping ref="MyGoal_2.result"/>
  </parameter>
  <body>new RealPlan_Planxy()</body>
  <trigger>
    <goal ref="MyGoal_1"/>
    <goal ref="MyGoal_2"/>
  </trigger>
</plan>
```

The plan body has to be completed by the developer, coding the functionalities of the ability represented by the plan (remember that here a plan represents the whole ability part of a capability, see Section 2.4.4). Initially, it contains only convenience code to access parameters and resources.Skeletons for abilities can be generated, using the tool presented in [Penserini et al., 2006a], following plan decomposition diagrams and UML 2.0 diagrams built at detailed design. See Section 4.3.4 for an automatic integration of abilities. For testing purposes, simple business logic can be written directly into the plan body.

¹ http://java.sun.com
4.3 Implementation

Goals & goal requests - delegation

Tropos goals are mapped directly to Jadex goals with the same name. By convention, all blanks in names of Tropos entities are converted to underscores, to prevent problems and to simplify name parsing at run-time. Goals have the same parameters than the Plans, which are linked to them. As discussed earlier, all goals are by default mapped to achieve goals without conditions. However, the developer can adapt them to his needs, adding conditions (e.g. creation conditions, achievement conditions,...) or changing the goal type, e.g. to maintainGoal. Using maintain-goals, it would be convenient to dispatch the goals in the agent's initial state and to remove the goal's request event and plan. Do not change non-root goals to maintain goals; they could get out of control because they are cyclic and do not end when the goal is achieved.

```xml
<achievegoal name="MyGoal">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>

<beliefset name="goals" class="TGoal">
  <fact>Components.createGoal("MyGoal", "OR")</fact>
  ...
</beliefset>
```

In addition to the <goals> section, goals are written to the belief base, along with their decomposition type. This is needed to navigate the goal hierarchy.

As defined in the mapping, if a goal has to be requested by the user (it was delegated) or by another agent, a request-plan (requestPlan_{goal-name}) is added to the ADF and triggered by a specific message-event. This event applies to all FIPA-REQUEST messages, whose content starts with the goal's name, followed by a blank and a parameter string. A single plan class (GoalRequestPlan.java) manages the requests of all goals. If needed, custom classes can be written. The GoalRequestPlan extends an abstract BaseGoalRequestPlan, implementing some methods that define the actions to perform on request failure and on success or failure of the requested goal. The plan body, in the base class, implements a request responder protocol.

```xml
<plan name="requestPlan_MyGoal">
  <body>new GoalRequestPlan("MyGoal")</body>
  <trigger>
    <messageevent ref="request_MyGoal"/>
  </trigger>
</plan>
```
<events>
  <!-- Specifies a request to achieve a goal (one per goal). -->
  <messageevent name="request_MyGoal" direction="receive" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.REQUEST</value>
    </parameter>
    <parameter name="content-start" class="String" direction="fixed">
      <value>"MyGoal "</value>
    </parameter>
  </messageevent>
  ...
</events>

**Softgoals & contribution**

Softgoals and contributions are mapped to the belief base. The first ones are stored, like goals, in a static hash-table in the `Components` class, to be retrieved by name. Contributions for a goal or plan can be retrieved using the predefined expression `query_contributions`. A list is returned, containing objects that deliver softgoal name, importance, and a value for the contribution.

```xml
<beliefset name="softgoals" class="TSoftgoal">
  <fact>Components.createSoftgoal("SoftGoal_1", 0.5)</fact>
</beliefset>

<beliefset name="contributions" class="TContrib">
  <fact>new TContrib("MyPlan", "SoftGoal_1", "+")</fact>
  ...
</beliefset>

<expression name="query_contributions">
  select $link
  from TContrib $link in $beliefbase.contributions
  where $link.get(0).equals($component)
  <parameter name="$component" class="String"/>
</expression>
```

Softgoal importance (and also resource values) can be changed at run-time by the user, sending a special FIPA message with performative `INFORM`. Its content should have one of the following formats:

- change {softgoal_name} {float_value}
- change {resource_name} {parameter}

**Means-end relation**

Means-end relations are mapped directly, adding to a plan definition a goal as trigger. The relation is added also to the belief base. The expression `query_ME_link` is used when the hierarchy has to be navigated (actually only for goal selection at OR-decomposition), to get a list of plans attached to a goal.
4.3 Implementation

At run-time, the selection of the best plan to execute is managed by Jadex meta-reasoning. A meta-goal \( \text{meta}_{\{\text{goal-name}\}} \) and a plan \( \text{metaPlan}_{\{\text{goal-name}\}} \) triggered by that goal, are defined in the ADF. Each meta-plan is associated to the \text{MetaPlan}_{\{\text{goal-name}\}} class that inherits the body method from the \text{AmetaPlan} class. This meta-plan overrides the predefined Jadex plan selection logic, implementing the OR-decomposition selection semantics discussed earlier (working with softgoal contributions), that include navigation of the goal decomposition tree down to the plans with the valuation of softgoal contributions at all levels. Obviously, applied to means-end relations, the algorithm does only check softgoal contributions of all applicable plans. All decisions are logged using Jadex facilities. However, each meta-plan can be customized independently.

**OR-decomposition**

Goal OR-decompositions are very similar to goal-plan means-end relationships. The only practical difference in the ADF is, that “real” plans are substituted by service plans,
called dispatchGoalPlan_{goal-name}, triggered by the OR-decomposed goal. The decomposition link is also added to the belief base to enable goal tree navigation.

```xml
<plan name="dispatchGoalPlan_MyGoal">
  <parameter name="param" class="String">
    <goalmapping ref="MyGoal.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="MyGoal.result"/>
  </parameter>
  <body>
    new DispatchGoalPlan("MyGoal")
  </body>
  <trigger>
    <goal ref="MyGoal"/>
  </trigger>
</plan>

<beliefset name="decomp" class="TLink">
  <fact>new TLink("MyGoal", "Goal_2", 1)</fact>
  <fact>new TLink("MyGoal", "Goal_3", 1)</fact>
</beliefset>
```

The Java class associated to dispatch-goal-plans is a general DispatchGoalPlan class. Its body method dispatches the subgoal given as parameter in the ADF and manages input and result parameters. Furthermore, it monitors goal achievement. Selection of the best subgoal to pursue is made by meta-reasoning, defining meta-goals and meta-plans that use the same algorithm as for means-end relationships. If the dispatched goal fails, the plan lets itself fail by calling `fail()`. Jadex manages this failure and passes the notice to the parent goal, where the selection process by meta-reasoning is re-initiated, excluding the failed plan and, therefore, the subgoal, too.

**AND-decomposition**

As defined in the mapping, an AND-decomposition is realized by triggering only a so-called ANDGoalPlan_{goal-name}. Its body method, inherited from BaseANDDispatchGoalPlan, dispatches the sub-plans.

```xml
<plan name="dispatchANDGoalPlan_MyGoal">
  <body>
    new ANDGoalPlan_MyGoal()
  </body>
  <trigger>
    <goal ref="MyGoal"/>
  </trigger>
</plan>
```

Initially sub-plans were hard-coded within the plan class. In the actual implementation, they are read from belief base. Since AND-decompositions have already to be annotated in the belief base, like other relations, to enable goal tree navigation, no further code has to be added to the XML file. In addition, the decomposition can be easily
modified by the developer and inconsistencies are avoided. Goal priorities can be annotated within the definition of the decomposition link, in the belief base. The ANDGoalPlan uses the database query query_link to retrieve a list of all subgoals, already ordered by priority. Goals with higher priority are dispatched first. If priorities are equal or not defined, random order is used.

Sub-plans are dispatched in sequence, waiting at each goal for successful achievement, before proceeding. At the first reported failure, the dispatching plan lets itself fail, this failure is propagated by Jadex to the parent goal.

'why'-dependency

Dependencies are mapped to the dependee's belief base. In addition, the dependee agent has to implement a requestPlan for the dependum goal, identical to the one employed for goal requests. Each goal's decomposition logic (in the associated plan) has to query the belief base for needed dependencies and request them to the dependee actors, using the standard FIPA-request protocol. In its first version, the implemented prototype deals only with dependencies associated to AND-decomposed goals, whose ANDDispatchPlan is suited to includes the logic needed. To enable dependencies for OR- and means-end decomposed goals, too, an additional plan type would be necessary. Moreover, the dependee agents are retrieved directly by their instance name. In a more general implementation, agents should register their class or role to the Directory Facilitator, where all instances of the needed agent class/role could be retrieved.
Resources & use/produce relationships

Resources are not mapped automatically for abovementioned reasons. Nevertheless, they can be inserted as facts into the belief base, modified by a user message with format “change {resource_name} {parameter}” and accessed in the real plans using the implemented methods useResource() and produceResource().

The provided plan and component library

To make possible the mapping from Tropos to Jadex ADF files, a library with extended Jadex Plan classes and some classes used for belief base entries, where necessary. All plan classes (Figure 4.9) are extensions of jadex.runtime.Plan and are used for the real Tropos plans, meta-plans and the needed service plans. The leaf classes – all nearly empty – are not contained in the library, but proper to the mapped agent and can be adapted by the developer.

Figure 4.9.: Class diagram of the plan classes used in the mapping. Aplan and AMetaPlan extend jadex.runtime.Plan.
4.3 Implementation

To map Tropos components and relationships to the object-oriented belief base and facilitate their querying and use, some simple classes were implemented (Figure 4.10). Goals and softgoals are stored as TGoal or TSoftgoal in static hash tables in the Components class. The different relationships are represented by objects that store the mapped information as Tuple and provide several methods to access this data. Some other methods are used by the belief base query expressions written to the ADF. All Java code can be found in Appendix B.

![Class diagram](image)

**Figure 4.10:** Class diagram displaying the classes used for belief base objects mapped from Tropos components.

### Architectural issues

Several problems emerge when dealing with agents at run-time, most of them are related to reliability, usability, security, and performance. The developed prototype is mainly used for a feasibility study, so security and performance are not addressed. Focusing reliability and failure handling, non-terminating subgoals or plans can be a real problem for the goal satisfaction process. The same problem arises at goal delegation to another agent, where the delegating agent needs to wait for an answer. For actions controlled by generated plans (dispatching of subgoals and waiting for response for goal 'why'-dependencies), this problem has been adequately resolved by adding a time-out...
(actually fixed to 15 seconds, changeable in the `APlan` class). Plans triggered by Jadex could be controlled by timer-based context conditions.

Another issue are repeated and parallel goal requests. If an achieve goal is repeatedly activated, in the current implementation, plans are executed for every request. To improve this behaviour, achievement conditions should be introduced. They are checked before and during the goal achievement process, and if they are true, the goal is already achieved and can be dropped.

### 4.3.3 The Code-Generator

The mapping from Tropos diagrams to Jadex agents can show its practical benefits especially with an automated mapping process. The Jadex code generation tool, called `t2x`, takes in input a Tropos model and writes a complete agent, ready to be executed on the Jadex platform. Code can be generated with “one-click”, enabling a fast and simple generation of Jadex agent structures, and facilitating the development of MAS also for non-expert programmers, by strongly reducing the coding effort.

The “*.tropos”-file, an XML-file with XMI format written by Taom4e, is parsed by a library provided by the Taom developers. The resulting model is accessed by a class that provides a navigation interface to the model parts of interest for the producer tool. Starting from a single agent, defined as input argument, the main class (`Producer.java`) analyses the decomposition of all root goals, navigating the model, and stores all information necessary for the mapping in an internal data structure with hash-tables and a goal tree structure (`AgentDefinition.java`). The navigation of the goal tree is terminated at each leaf, where a plan is encountered; this plan would be the starting point for a capability (see the next section).

As soon as the navigation is completed, the code mapping process (mainly in `AgentWriter.java`) is carried out. The ADF file is produced, replacing placeholders in template XML files, with patterns filled during the writing process. First, Softgoals are written to the belief base, then the hash-tables are read sequentially and specific patterns (all stored as XML files) for the different types of goals and plans are filled with the extracted data. All encountered relationships are added to the belief-base and, for every real plan and service plan, proper Java files are written. These files are generated from plan skeletons, where only file-name, package name and class-name have to be changed, because all the mapping logic is gathered into the ADF file. This architecture gives the
possibility to make some changes to the ADF file or to the patterns without to change the application code and keeps code clean. At the end, all Java files, which are an integral part of the mapped agent and do not belong on the mapping itself, are copied, such as the general plans for dispatching goals and managing requests (dispatchGoalPlan and requestPlan). The generated code is completed by batch files, to launch the Jadex platform and the generated agent.

Class diagrams and the whole code for the code generation tool t2x can be found in Appendix C.

4.3.4 Integration of the abilities part

The agents generated with the t2x tool are executable, but they contain only the knowledge level, the “reasoning part” of the agent. The business logic for the implemented “real” plans can be generated by the tool presented in [Penserini et al., 2006b]. This tool prototype, mainly written by the students Barbara Tomasi and Loris Delpero, produces for each ability extracted from a Tropos model a Jade agent with a finite-state-machine behaviour, containing the mapped activities. To be integrable with the Jadex agent, the generation of these abilities has to be adapted. Each ability is put into a single finite state machine, and no more correlated to the executing agent for passing from one state to the other. In this way, Jade agents can execute more than one ability.

The generated abilities have to be executable from a Jadex plan. We decided not to start the Jade behaviours directly from a Jadex plan, but to implement an agent, which manages the abilities available for an agent and accepts requests for ability execution. This technique has various advantages: no problems with message handling between Jadex and the Jade behaviours arise, the code is modular, and a standard protocol can be used to make requests and manage failures.

The implemented CapabilitiesAgent is a simple service-providing Jade agent without reasoning ability. It has to be started with the Jadex agent and listens for FIPA-request messages, containing the name of an ability and some parameters. The respective ability class (a Jade behaviour) is searched in an ability repository on the file system and the behaviour is activated. Success or failure is eventually returned to the requesting agent.
Requests are made from the real plans modelled in Jadex, using the same protocol as for goal dependencies. A time-out prevents blocking; success and failures are passed to the higher levels of the goal hierarchy.

This system could be used in future, for a feasibility study on run-time extension of the available capabilities. To realize this, only the Jadex agent would have to be adapted, to be able to add new means to a goal at run-time. The new abilities are ready to be executed, as soon as they are placed into the agent's ability repository.
5 Preliminary Experiments

This Chapter presents one of the experiments carried out with the automatic code generation tool presented in the last chapter. A simplified development process is put into practice, from requirements analysis to the execution. Special attention is dedicated to the knowledge level engineering at architectural design and implementation phases. However, the example does not deal with plan decomposition, modelling and implementation of concrete agent abilities, that are supposed to be already implemented and collected in a capability repository. The whole generated ADF XML file and the Java code of some generated plan classes can be found in Appendix A. While, Tropos diagrams shown in the following figures are drawn with Taom4e and are the input for the code generation tool.

For the sake of simplicity, we chose to develop a system that gives assistance to students and teachers. The focus is given on the design of an important part of such a system, a word searching agent. This agent has been analysed according to its principal goals in order to prepare the input for the proposed mapping into a Jadex agent.

5.1 Domain and System Requirements

To pass an exam, students have to deliver some written homework, autonomously investigating on some facts. The “SearchSystem” provides an extended encyclopaedia service, to lookup word descriptions. On the other side, the teacher that has to correct the work, needs to control somehow if the work was written by the student itself or
Preliminary Experiments

(partially) copied. Here the “SearchSystem” provides a payment service that finds out, in a work text, parts literally copied from Internet or from an encyclopaedia. Figure 5.1 shows a diagram of the system late requirements, drawn with Taom4e, according to a previous domain analysis. In particular, each student delegates the goal “find word description” to the software system, hoping to obtain reliable results. Teachers delegate the goal “find copied text”, providing the student's works, and demand reliable results, without to overload their budget that is known to be anyway too low.

5.2 Architectural Design

Now in Taom4e a new architectural diagram is built to detail the system-to-be (Figure 5.2). The “SearchSystem” is decomposed into sub-actors, to which the fulfilment of goals and softgoals is delegated. The sub-actors identified for this small multi-agent system are SearchActor, ExamParser and WebServer. These shall represent the software agents to implement. To implement a system, where these actors represent not agents, but roles, Section 4.2.5 can be consulted, but the underlying principles do not change. We focus particularly on goal analysis for the primary actor, the SearchActor. In it, the two hard goals delegated to the system are analysed, defining an engine that can search requested words in the Internet or in different encyclopaedias, and the ability to check text for matchings.

The goal “find word description”, delegated from the student, is OR-decomposed into the two goals “find in Internet” and “find in encyclopaedia”. To perform an Internet search, the actor first depends on the “WebServer” actor, to find related web-pages. The “WebServer” is not detailed, for completeness we define a plan “googleSearch” that satisfies the delegated goal. Next, the goal “find in Internet”, is AND-decomposed. First, pages need to be filtered and then the results need to be parsed. These two goals are satisfied by the two plans “ContentFilter” and “parser”, respectively. With softgoal contributions, we define that Internet search gives cheap, but not very reliable results. The goal “find in encyclopaedia” has two plans in means-end that can be executed to achieve it. One plan, to search in the Encyclopaedia Britannica, is very reliable, but costly. The second plan, to search in the Wikipedia, which is free and gives good results, can be queried.

The goal “find copied text”, delegated from the teacher actor, is AND-decomposed into the already analysed goal “find word description” and a goal named “find match-
5.2 Architectural Design

ing”, achievable executing the plan “MatchText”. Moreover, the root goal depends on the “ExamParser” actor, to parse the text that has to be given by the teacher that requests the service. This actor has only one plan in means-end to the delegated goal.

Moreover, the root goal depends on the “ExamParser” actor, to parse the text that has to be given by the teacher that requests the service. This actor has only one plan in means-end to the delegated goal.

Until now, the model does not deal with annotations correlated to the contribution values. Therefore, the goal sequence at each AND-decomposition has to be defined manually at the implementation phase.

Furthermore, we have to stress that, for the aim of this thesis no plan decomposition is shown. The plans modelled here can be seen as very simple tasks or as the “starting point” for more complex abilities modelled at Detailed Design phase. In this example only the knowledge level modelling is showed, thus, following the proposed process we can go directly to the implementation phase.
5.3 Implementation

The implementation of the reasoning part of the system agents in the system can now be carried out basing on the previously drawn Tropos model file containing the agents with their goal decomposition diagrams (Figure 5.2). The code generator tool has to be executed with two parameters: the Tropos model file (*.tropos) containing the Taom4e diagram and the name of the agent from the model that has to be implemented. Using some facilities provided by Eclipse, these arguments can be provided simply by selecting the .tropos-file in the Navigator window and setting the agent in the appearing dialogue box. In its next version, the tool should be fully integrated with the Taom4e plug-in.

The code generation has to be executed for all three agents in the system and generates for every agent a directory, with the relative ADF and Java plan files. The whole ADF file generated for the SearchActor, 450 lines long, can be found in Appendix A. The agent structure written in the ADF is the result of the mapping from Tropos to Ja dex concepts as defined in Section 4.2 and its graphical representation would be conceptually an extension of Figure 4.8, with request plans, dispatch plans, meta-goals, etc. It is ready to be executed, only priorities for AND-decomposed goals have to be set manually in the belief base, like shown in the following lines:

```xml
<fact>new TLink("find_in_Internet", "filter_pages", 2)</fact>
<fact>new TLink("find_in_Internet", "parse_result", 1)</fact>
```

The following XML code shows the ADF file generated for the small WebServer agent, reduced to the most significant parts.

```xml
<agent name="WebServer" package="t2x.generated.WebServer">
  <goals>
    <achievegoal name="find_webpages">...
    <metagoal name="meta_find_webpages">
      <trigger> <goal ref="find_webpages"/> </trigger>
    </metagoal>
  </goals>
  <plans>
    <plan name="requestPlan_find_webpages">
      <body>new GoalRequestPlan("find_webpages")</body>
      <trigger><messageevent ref="request_find_webpages"/></trigger>
    </plan>
    <plan name="metaPlan_find_webpages">
      <body>new MetaPlan_find_webpages()</body>
      <trigger> <goal ref="meta_find_webpages"/> </trigger>
    </plan>
  </plans>
</agent>
```
5.3 Implementation

The WebServer contains only one hard goal (line 3) and one real plan (lines 17 to 20), that are also defined in the belief base (not shown here). The message-event request_find_webpages, on line 23, is needed as trigger for the requestPlan (lines 9 to 12), to accept goal satisfaction requests from the SearchActor. The metagoal associated to the goal and its metaplan are shown, even if they are not really needed in this simple example, because there is only one plan in means-end to the goal.

5.4 Execution

To start and test the generated agents, we first set up the Jadex platform and start a test agent which can simulate a user by sending request messages and waiting for responses. Our agents can then be started traditionally through the Jadex Control Center, from command line or with the implemented Eclipse run-configuration, selecting an agent's directory in the Navigator and clicking on “run selected agent”.

5.4.1 Testing the multi-agent system

If all three system agents are started, we can test them, sending a request to the SearchActor. We make a request to find the description of the word “Tropos”, sending from the test agent the FIPA request “find_word_description Tropos” to the SearchActor. The SearchActor receives the request and a message-event is thrown, which triggers the request-plan associated to the goal “find_word_description”. The request-plan triggers the associated goal, and the goal achievement process starts.

We can easily see, that the plan “Wikipedia” gives the highest contribution to the two softgoals. Therefore, the agent tries to achieve the goal “find in encyclopaedia” and subsequently to execute the plan “Wikipedia”. To make the goal achievement process more interesting, we suppose that the keyword “Tropos” cannot be found in the Wikipedia, nor in the Encyclopaedia Britannica, therefore these goals fail (simply by calling the fail() method in the related Java plans). So if “Wikipedia” fails, the only plan able to achieve the subgoal “find in encyclopaedia” remains “EBritannica”, and this plan
fails, too. It follows that the subgoal fails, and the agent has to dispatch the subgoal “find in Internet” to achieve the requested goal. This goal has, as we have seen previously, a dependency to the WebServer agent and two subgoals that have both to be successfully achieved, to achieve the requested goal.

**Graphical representation**

Goals, Jadex plans and messages activated during the goal achievement process can be represented graphically in the Jadex Introspector. Starting from a small agent symbol, it builds in real-time a graph that represents the hierarchy of dispatched goals, plans triggered by them, subgoals dispatched from those plans, and so on. Additionally, outgoing and incoming messages are shown as small arrows. Unfortunately, activated meta-goals with their plans are not associated to the related goals, but directly to the agent. Figure 5.3 shows the part of the Tropos goal decomposition diagram related to the goal “find_word_description” and the “molecule-like” graph shown in the Jadex Introspector, at completion of the achievement process.

![Graphical representation of Jadex Introspector](image)

Figure 5.3: Goal decomposition for “find word description” and graphical run-time representation of the agent’s internal structure, generated by the Jadex Introspector tool.

The graph shows first the agent and the incoming request (on the right side). Through a request-plan, the OR-decomposed goal “find word description” is dispatched (at the centre of the graph). Going downwards from this goal, we can see a dispatch-goal plan and the symbol associated to the goal “find in encyclopaedia”. From here, two real plans are dispatched in sequence, both fail as previously defined. This failure is returned to the OR-decomposed goal, that now tries to achieve the other sub-
5.4 Execution

goal in OR-relation, “find in Internet”, and we can see the graph evolve to the right side. The goal “find in Internet” triggers a AND-decomposition-plan, which manages achievement of its subgoals and the fulfilment of the dependency with the WebServer (the small arrow on the left side). Now all plans succeed and this information is passed back (together with the parameters that contain the result of the search process) to the initial request plan, that finally sends an Inform to the user, which made the request.

Communication

Figure 5.4 shows the inter-agent communication following to a request for achievement of the goal “find copied text”. All inter-agent communication in the system relies on the FIPA-Request protocol, that defines agree and inform response messages. The protocol also allows different failure messages, like failure, refuse and not-understood. If for an agent an ability repository, managed by an agent like proposed in Section 4.3.4, is available, communication between them is carried out using the FIPA-Request protocol, too.

The following two figures, Figure 5.5 and Figure 5.6, are screen-shots from the Jadex Introspector tool, showing the real-time evolution of the goal achievement process of the goal “find copied text”. Frame A shows the three system agents; WebServer on the left,
ExamParser on the right and SearchActor on the lower side that already received the goal request and executed the request-plan. In B the request to the ExamParser was already made, that fulfilled the dependency and returned agree and inform messages. In C, the goal achievement process proceeds locally, activating goals and plans.

![Figure 5.5: Expansion of agent goal trees during the achievement process for the goal “find copied text”.

On the upper side of frame D, the goal “find in encyclopaedia” triggers the two associated plans, that both fail. Therefore, the parent goal dispatches, in frame E, the goal “find in Internet”, which sends a request (arrow at the upper right side) to the WebServer agent. The request is fulfilled and an Inform-message is sent to the SearchAgent, that can now continue to achieve the AND-decomposed goal (frame F), dispatching subgoals and executing the plans in means-end to them (on the left side of the graph). Results are passed back to the root plan that managed the initial request. Finally it can respond to the user's request with an Inform-message.

![Figure 5.6: Completion of the goal achievement process started in Figure 5.5 (screen-shots from the Jadex Introspector).

The tools utilized are very helpful to follow the goal achievement process. It can be seen, that the agent follows the proposed mapping and behaves basing on the semantics defined in Section 3.3.2. Now it is up to the developer to change the behaviour of the generated agent's reasoning part to his needs and to implement the abilities.
6 Conclusions

In this thesis, knowledge-related information, retrieved by designing a system and following an agent-oriented software engineering methodology, has been explored to improve the reasoning behaviour of agent-based applications. The AOSE methodology TROPOS moves important parts of requirements analysis and system design from a software-centric level to a knowledge level, analysing stakeholders' goals, delegating and decomposing them. These parts have not been considered in a recent improvement of the methodology [Penserini et al., 2006a], which mainly concerns with the operative part of system functionalities and so lacks in to cope with important properties such as adaptability and failure handling.

This thesis proposed and described a way to effectively deal with the mapping between the agent's knowledge part (modelled by TROPOS diagrams) and implemented BDI agent structures. Conscious at run-time about the modelled knowledge level information, agents are able to react to failures and to changes in the environment in a way that meets the designed requirements. The proposed approach allows the developer to automatically generate BDI agents that are aware of this information, selecting subgoals to pursue with respect to proposed design time semantics (by means of goal decompositions and inter-agent dependencies). The implemented agents are also able to deal with non-deterministic selections in the goal achievement process, by the use of contributions of plans and goals to quality-of-service oriented softgoals.

We defined concrete semantics for a diminished set of knowledge level-related TROPOS structures and relationships. From this basis we developed a mapping to BDI-related agent-oriented concepts, focusing on an implementation of Jadex agents. A specific implementation of these concepts is the basis for a prototype tool that automatically generates BDI agent code, starting from TROPOS models designed with the tool TAOM4E on the Eclipse platform. The BDI agent code consists of XML Agent Description Files (ADF) together with different Java files, executable on the Jadex platform.

We aimed at the generation of code that is easy to understand, to adapt, and to extend. The implemented agents use as much as possible the reasoning mechanisms
available in Jadex to deliberate on and dispatch goals, while dealing with failures and reasoning on a hierarchical goal structure with dependencies to other agents.

Generated agent structures have to be completed coding the business logic for the agent abilities. If a capability repository, created as proposed in [Penserini et al., 2006a], is available, it can be easily integrated. Possibly some additional information, like temporal priorities, constraints or inhibitions among goals, have to be added to the agent reasoning part. In order to enrich the TROPOS model by such information, we are investigating the use of some diagram annotations.

Finally, we made some case studies to validate the approach and the usability of the generated code. The reached results are very promising for their simplicity and their effective use of the underlying BDI Agent technology. Runnable agent structures can be generated from a TROPOS goal diagram with “one click”, strongly reducing the coding effort. Failure monitoring and recovery are managed by the agents, exploring modelled alternatives. The direct mapping of knowledge level information with runtime instances of software agents facilitates tracking of decisions made on run-time down to the requirements engineering phases and vice versa. In an iterative development process that includes automatic generation and execution of designed diagrams, it would be possible to map changes on the design directly to the implementation, while agents' run-time decisions can be considered to refine the architectural design model.

The mapping defined in this thesis has to be further extended to cover more of the possibilities given by the TROPOS metamodel and to refine the goal selection process, including preconditions, resources and contributions between goals. Moreover, the applicability of such a straight mapping and automatic code generation to real case studies has to be examined. It has to be evaluated whether more knowledge level information can be extracted from TROPOS diagrams without reducing their expressiveness, and which extensions to the design process would be useful to improve the versatility of generated systems, including proactive and learning agents. Furthermore, the implemented prototype has to be fully integrated with the capability generation process and with the TAOM4E Eclipse plug-in, possibly linking the generated code directly to the entities of the diagrams. Last but not least, we need to built up a formalisation in some BDI logics in order to make the mapping process towards BDI concepts precise and unambiguous. The work on the topics of this thesis and further improvement of the realized tools will be continued during the author's doctorate studies.
Conclusioni

In questa tesi sono state esaminate le informazioni a livello di conoscenza contenute in un modello di sistema ad agenti, sviluppato usando una metodologia di ingegneria del software orientata agli agenti, con il fine di facilitare il processo di implementazione e di migliorare il comportamento di applcazioni ad agenti.

Nella metodologia Tropos, una parte importante che caratterizza progettazione e sviluppo del sistema è stata spostata da un livello basato su concetti software tradizionali ad un livello di conoscenza (knowledge level), analizzando goal, delegando e decomponendoli. Però questa parte non è stata considerata nella recente estensione della metodologia [Penserini et al., 2006a], che si occupa soprattutto della parte operativa delle funzionalità del sistema.

Questa tesi propone e descrive un mapping della parte di knowledge level modellata con TROPOS verso l'implementazione di strutture ad agenti basate su un'architettura BDI. Un'agente che ha queste conoscenze, è capace di reagire a fallimenti ed a cambiamenti dell'ambiente in modo conforme ai requisiti modellati. L'approccio proposto consente allo sviluppatore di generare automaticamente degli agenti BDI, che sono capaci di selezionare i sotto-goal da perseguire rispettando il modello disegnato (per mezzo di decomposizioni di goal e dipendenze tra agenti).

Abbiamo definito una semantica concreta per un set ridotto di strutture e relazioni TROPOS, relative al knowledge level. Per questo set è stato elaborato un mapping verso concetti relativi ad un'architettura di agenti BDI, mirando ad un'implementazione di agenti Jadex. Questa implementazione è la base per un'applicazione prototipo che genera automaticamente codice ad agenti, partendo da modelli TROPOS disegnati con il tool TAOM4E. Il codice generato consiste di un Agent Definition File in formato XML, eseguibile sulla piattaforma Jadex, completato di diversi file Java. Gli agenti implementati sono anche capaci di affrontare selezioni non-deterministiche nel processo di raggiungimento di un goal, esaminando contribuzioni di piani e goal verso softgoal che descrivono qualità del servizio.

Uno degli obiettivi è stato di generare (da subito) codice eseguibile organizzato in strutture dati semplici da adattare e da estendere. Quindi, gli agenti implementati usano
Conclusioni

il più possibile i meccanismi di ragionamento presenti in Jadex, per deliberare ed attivare goal e per trattare fallimenti. Ovviamente, il software deve essere completato implementando le funzionalità concrete dell'agente. Se sono disponibili delle capability, implementate seguendo la proposta di [Penserini et al., 2006a], queste possono essere facilmente integrate. Eventualmente è necessaria l'aggiunta di informazioni addizionali a livello di conoscenza, come priorità temporali, costrizioni o inibizioni tra goal. Stiamo esaminando la possibilità di aggiungere annotazioni a diagrammi per arricchire il modello TROPOS di queste informazioni.

Infine abbiamo sviluppato alcuni casi di studio, per validare l'applicabilità dell'approccio proposto e la versatilità del codice generato. Strutture eseguibili di agenti possono essere generate da un diagramma TROPOS senza necessità di interazione con lo sviluppatore, riducendo di molto la quantità di codice da scrivere. Il monitoraggio di fallimenti e loro recupero sono prerogativa dell'agente implementato, esplorando le alternative modellate. Il mapping diretto di informazioni dal livello di conoscenza alle reali istanze di agenti può facilitare eventuali lavori futuri di tracciabilità di cambiamenti che avvengono a livello di run-time verso il design: per esempio aggiornare i modelli concettuali con nuova conoscenza acquisita dall'agente durante interazioni con altri agenti. In altre parole, in un processo iterativo che include la generazione automatica e l'esecuzione dei diagrammi modellati, sarebbe possibile mappare decisioni prese dagli agenti a run-time al design-time, per perfezionare il design.

Il mapping definito in questa tesi dovrà essere ulteriormente esteso per coprire più concetti del modello TROPOS e per perfezionare il processo di selezione di goal, includendo precondizioni, risorse e contribuzioni tra goal. Inoltre dovrà essere valutata l'applicabilità del mapping definito e della generazione automatica di codice a casi di studio reali. Dovremmo esaminare, se è possibile estrarre ulteriori informazioni a livello di conoscenza da diagrammi TROPOS, o se è opportuno definire delle estensioni al processo di sviluppo, per poter generare agenti pro-attivi ed agenti abili ad aumentare la loro conoscenza durante l'esecuzione. Il prototipo implementato sarà completamente integrato con il processo di generazione delle capability e con il tool TAOM4E. Infine vorremmo fornire una formalizzazione in una logica BDI, per avere una base precisa ed univoca per il processo di implementazione.
I contenuti di questa tesi sono stati elaborati durante uno stage al centro di ricerca ITC Irst Trento, nel dipartimento Sistemi per il Ragionamento Automatico (SRA), da febbraio a luglio 2006. Vorrei ringraziare per primo Dr. Loris Penserini, il mio tutor durante il periodo di stage e correlatore di questa tesi. Era lui che mi ha suggerito di fare delle prime esperienze di ricerca all'Irst, proponendomi di lavorare nell'ambito della programmazione ad agenti, e che mi ha seguito ed assistito. Loris ha saputo coinvolgermi nelle attività di ricerca, facendomi partecipare attivamente a riunioni e discussioni ed ascoltando le mie opinioni. Vorrei ringraziare lui ed anche il secondo correlatore, Dr. Angelo Susi, per le idee e gli aiuti che mi hanno dato durante il periodo di stage e per la correzione accurata della tesi, e specialmente il mio relatore ufficiale, Dr. Paolo Giorgini. Inoltre ringrazio i responsabili del dipartimento SRA, soprattutto Dr. Anna Perini, per darmi la possibilità di continuare gli studi per il dottorato di ricerca. Grazie anche a tutti i collaboratori, innanzitutto Alberto, Cu e Davide Bertolini, che mi ha dato preziosi aiuti su Taom.

Vor allem aber möchte ich meinen Eltern danken, die mir dieses Studium ermöglicht und mich während der gesamten Studienzeit mit aller Kraft unterstützt haben, und ganz besonders auch meiner Freundin Simone, für die Geduld, die sie für mich diesen Sommer über aufbringen musste.
Appendix A:
Example ADF XML file and plans

ADF file for SearchActor

<!--
Automatic implementation of Tropos goal diagrams to Jadex.
"T2X" Author: Mirko Morandini, ITC-irst / University of Trento (I), 2006
Auto-generated file, please change it to your needs.
-->
<agent xmlns="http://jadex.sourceforge.net/jadex"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jadex="http://jadex.sourceforge.net/jadex"
xsi:schemaLocation="http://jadex.sourceforge.net/jadex http://jadex.sourceforge.net/jadex-0.94.xsd"
name="SearchActor"
package="t2x.generated.SearchActor"

<imports>
<import>jadex.util.*</import>
<import>jadex.adapter.fipa.*</import>
<import>jadex.runtime.*</import>
<import>java.util.logging.*</import>
<import>t2x.util.plans.*</import>
<import>t2x.util.components.*</import>
</imports>

<beliefs>
<!--The belief contains the tropos hierarchy as facts. -->
<beliefset name="goals" class="TGoal">
  <fact>Components.createGoal("filter_pages", "ME")</fact>
  <fact>Components.createGoal("find_in_internet", "AND")</fact>
  <fact>Components.createGoal("find_matching", "ME")</fact>
  <fact>Components.createGoal("find_in_encyclopaedia", "ME")</fact>
  <fact>Components.createGoal("parse_result", "ME")</fact>
  <fact>Components.createGoal("find_word_description", "OR")</fact>
</beliefset>

<beliefset name="softgoals" class="TSoftgoal">
  <fact>Components.createSoftgoal("reliable_results", 0.5)</fact>
  <fact>Components.createSoftgoal("minimize_cost", 0.5)</fact>
</beliefset>

<beliefset name="decomp" class="TLink">
  <fact>new TLink("find_word_description", "find_in_internet")</fact>
  <fact>new TLink("filter_pages", 2)</fact>
  <fact>new TLink("parse_result", 1)</fact>
  <fact>new TLink("find_matching", 1)</fact>
  <fact>new TLink("find_in_encyclopaedia")</fact>
</beliefset>

<beliefset name="meansend" class="TLink">
  <fact>new TLink("filter_pages", "contentFilter")</fact>
  <fact>new TLink("find_in_encyclopaedia", "EBritannica")</fact>
  <fact>new TLink("find_matching", "matchText")</fact>
  <fact>new TLink("find_in_encyclopaedia", "wikipedia")</fact>
  <fact>new TLink("parse_result", "parser")</fact>
</beliefset>

<beliefset name="contributions" class="TContrib">
  <fact>new TContrib("find_in_internet", "reliable_results", "+")</fact>
  <fact>new TContrib("find_in_internet", "minimize_cost", "+")</fact>
  <fact>new TContrib("EBritannica", "reliable_results", "+")</fact>
  <fact>new TContrib("EBritannica", "minimize_cost", "+")</fact>
  <fact>new TContrib("wikipedia", "reliable_results", "+")</fact>
  <fact>new TContrib("wikipedia", "minimize_cost", "+")</fact>
</beliefset>

<beliefset name="dependencies" class="TDependency">
<!--TDependency: 'why'-Goal (must be AND-decomposed), Dependum, Dependee --
  <fact>new TDependency("find_in_internet", "find_webpages", "MyWebServer")</fact>
</beliefset>
Appendix A: Example ADF XML file and plans

```xml
<fact>new TDependency("find_copied_text", "parse_exam", "MyExamParser")</fact>
</beliefset>
<![--$RESOURCES-->]
</beliefset>
<goals>
<!-- default values: metalevelreasoning enabled, exclude when tried. -->
<achievegoal name="filter_pages">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="find_in_Internet">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="find_copied_text">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="find_matching">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="find_in_encyclopaedia">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="parse_result">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="find_word_description">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>
<achievegoal name="meta_filter_pages">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="filter_pages"/>
  </trigger>
</metagoal>
<metagoal name="meta_find_matching">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="find_matching"/>
  </trigger>
</metagoal>
<metagoal name="meta_find_in_encyclopaedia">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="find_in_encyclopaedia"/>
  </trigger>
</metagoal>
<metagoal name="meta_parse_result">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="parse_result"/>
  </trigger>
</metagoal>
<metagoal name="meta_find_word_description">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="find_word_description"/>
  </trigger>
</metagoal>
</goals>
<plans>
<![-- Initial plans for handling message requests, creating appropriate subgoals. They use waitqueues to store message events that arrived during the plan was busy. -->]
<plan name="requestPlan_find_copied_text">
  <body>new GoalRequestPlan("find_copied_text")</body>
  <trigger>
    new GoalRequestPlan("find_copied_text")
  </trigger>
</plan>
```
<messageevent ref="request_find_copied_text"/>
</trigger>
</plan>

<plan name="requestPlan_find_word_description">
<body>
<GoalRequestPlan("find_word_description")></body>
<trigger>
<messageevent ref="request_find_word_description"/>
</trigger>
</plan>

<plan name="informChangePlan">
<body>
<InformChangePlan()></body>
<trigger>
<messageevent ref="inform_ChangeEnv"/>
</trigger>
</plan>

<!-- Plans triggered by a parent goal, used to dispatch a child goal. -->
<plan name="dispatchGoalPlan_find_in_Internet">
<parameter name="param" class="String">
<goalmapping ref="find_word_description.param"/>
</parameter>
<parameter name="result" class="String">
<goalmapping ref="find_word_description.result"/>
</parameter>
<body>
<DispatchGoalPlan("find_in_Internet")></body>
<trigger>
<goal ref="find_word_description"/>
</trigger>
</plan>

<plan name="dispatchGoalPlan_find_in_encyclopaedia">
<parameter name="param" class="String">
<goalmapping ref="find_word_description.param"/>
</parameter>
<parameter name="result" class="String">
<goalmapping ref="find_word_description.result"/>
</parameter>
<body>
<DispatchGoalPlan("find_in_encyclopaedia")></body>
<trigger>
<goal ref="find_word_description"/>
</trigger>
</plan>

<!-- Plans associated to an AND-decomposed goal, used to dispatch all subgoals in sequence. -->
<plan name="dispatchANDGoalPlan_find_in_Internet">
<parameter name="param" class="String">
<goalmapping ref="find_internet.param"/>
</parameter>
<parameter name="result" class="String">
<goalmapping ref="find_internet.result"/>
</parameter>
<body>
<ANDGoalPlan_find_in_Internet()></body>
<trigger>
<goal ref="find_in_Internet"/>
</trigger>
</plan>

<plan name="dispatchANDGoalPlan_find_copied_text">
<parameter name="param" class="String">
<goalmapping ref="find_copied_text.param"/>
</parameter>
<parameter name="result" class="String">
<goalmapping ref="find_copied_text.result"/>
</parameter>
<body>
<ANDGoalPlan_find_copied_text()></body>
<trigger>
<goal ref="find_copied_text"/>
</trigger>
</plan>

<!-- Meta-Plans associated to their Metagoals, used to chose between alternative Plans (and so between goals)-->
<plan name="metaPlan_filter_pages">
<parameterset name="applicables" class="ICandidateInfo">
<goalmapping ref="meta_filter_pages.applicables"/>
</parameterset>
<parameterset name="result" class="ICandidateInfo" direction="out">
<goalmapping ref="meta_filter_pages.result"/>
</parameterset>
</plan>
<body>new MetaPlan_filter_pages()</body>
<trigger>
  <goal ref="meta_filter_pages"/>
</trigger>
</plan>

<plan name="metaPlan_find_matching">
  <parameterset name="applicables" class="ICandidateInfo">
    <goalmapping ref="meta_find_matching.applicables"/>
  </parameterset>
  <parameterset name="result" class="ICandidateInfo" direction="out">
    <goalmapping ref="meta_find_matching.result"/>
  </parameterset>
  <body>new MetaPlan_find_matching()</body>
<trigger>
  <goal ref="meta_find_matching"/>
</trigger>
</plan>

<plan name="metaPlan_find_in_encyclopaedia">
  <parameterset name="applicables" class="ICandidateInfo">
    <goalmapping ref="meta_find_in_encyclopaedia.applicables"/>
  </parameterset>
  <parameterset name="result" class="ICandidateInfo" direction="out">
    <goalmapping ref="meta_find_in_encyclopaedia.result"/>
  </parameterset>
  <body>new MetaPlan_find_in_encyclopaedia()</body>
<trigger>
  <goal ref="meta_find_in_encyclopaedia"/>
</trigger>
</plan>

<plan name="metaPlan_parse_result">
  <parameterset name="applicables" class="ICandidateInfo">
    <goalmapping ref="meta_parse_result.applicables"/>
  </parameterset>
  <parameterset name="result" class="ICandidateInfo" direction="out">
    <goalmapping ref="meta_parse_result.result"/>
  </parameterset>
  <body>new MetaPlan_parse_result()</body>
<trigger>
  <goal ref="meta_parse_result"/>
</trigger>
</plan>

<plan name="metaPlan_find_word_description">
  <parameterset name="applicables" class="ICandidateInfo">
    <goalmapping ref="meta_find_word_description.applicables"/>
  </parameterset>
  <parameterset name="result" class="ICandidateInfo" direction="out">
    <goalmapping ref="meta_find_word_description.result"/>
  </parameterset>
  <body>new MetaPlan_find_word_description()</body>
<trigger>
  <goal ref="meta_find_word_description"/>
</trigger>
</plan>

<!-- Real Plans that hold the activity part of a capability and "do" the requested things -->

<plan name="realPlan_contentFilter">
  <parameterset name="applicables" class="String">
    <goalmapping ref="filter_pages.param"/>
  </parameterset>
  <parameterset name="result" class="String">
    <goalmapping ref="filter_pages.result"/>
  </parameterset>
  <body>new RealPlan_contentFilter()</body>
<trigger>
  <goal ref="filter_pages"/>
</trigger>
</plan>

<plan name="realPlan_EBritannica">
  <parameterset name="applicables" class="String">
    <goalmapping ref="find_in_encyclopaedia.param"/>
  </parameterset>
  <parameterset name="result" class="String">
    <goalmapping ref="find_in_encyclopaedia.result"/>
  </parameterset>
  <body>new RealPlan_EBritannica()</body>
<trigger>
  <goal ref="find_in_encyclopaedia"/>
</trigger>
</plan>
<plan name="realPlan_matchText">
  <parameter name="param" class="String">
    <goalmapping ref="find_matching.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="find_matching.result"/>
  </parameter>
  <body>
    new RealPlan_matchText()
  </body>
  <trigger>
    <goal ref="find_matching"/>
  </trigger>
</plan>

<plan name="realPlan_Wikipedia">
  <parameter name="param" class="String">
    <goalmapping ref="find_in_encyclopaedia.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="find_in_encyclopaedia.result"/>
  </parameter>
  <body>
    new RealPlan_Wikipedia()
  </body>
  <trigger>
    <goal ref="find_in_encyclopaedia"/>
  </trigger>
</plan>

<plan name="realPlan_parser">
  <parameter name="param" class="String">
    <goalmapping ref="parse_result.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="parse_result.result"/>
  </parameter>
  <body>
    new RealPlan_parser()
  </body>
  <trigger>
    <goal ref="parse_result"/>
  </trigger>
</plan>

</plans>

<events>
<!-- Specifies a request to achieve a goal (one per goal). -->
<messageevent name="request_find_copied_text" direction="receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.REQUEST</value>
  </parameter>
  <parameter name="content-start" class="String" direction="fixed">
    <value>find_copied_text </value>
  </parameter>
</messageevent>

<messageevent name="request_find_word_description" direction="receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.REQUEST</value>
  </parameter>
  <parameter name="content-start" class="String" direction="fixed">
    <value>find_word_description </value>
  </parameter>
</messageevent>

<!-- messages for informs on environment changes-->
<messageevent name="inform_ChangeEnv" direction="receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.INFORM</value>
  </parameter>
  <parameter name="content-start" class="String" direction="fixed">
    <value>change </value>
  </parameter>
</messageevent>

<!--FIPA-messages needed to communicate with other agents-->
<messageevent name="request" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.REQUEST</value>
  </parameter>
  <parameter name="reply-with" class="String">
    <value>SFipa.createUniqueId($scope.getAgentName())</value>
  </parameter>
</messageevent>
</events>
Appendix A: Example ADF XML file and plans

<!--default FIPA-messages needed to communicate with other agents-->
<messageevent name="inform" direction="send_receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value> SFipa.INFORM </value>
  </parameter>
</messageevent>
<messageevent name="agree" direction="send_receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value> SFipa.AGREE </value>
  </parameter>
</messageevent>
<messageevent name="failure" direction="send_receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value> SFipa.FAILURE </value>
  </parameter>
</messageevent>
<messageevent name="n_u" direction="send_receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value> SFipa.NOT_UNDERSTOOD </value>
  </parameter>
</messageevent>
<messageevent name="refuse" direction="send_receive" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value> SFipa.REFUSE </value>
  </parameter>
</messageevent>
<expressions>
<!--All expressions are not changed during the automatic code generation-->  
<!--This query selects the first matching entry from the dictionary,  
whereby the parameter $word is compared to the first element of  
a belief set tuple. -->
<expression name="query_link">
  select $link.getGoal(1) 
  from TLink $link in $beliefbase.decomp 
  where $link.get(0).equals($component) 
  order by $link.getPriority() desc 
  <parameter name="$component" class="String"/>
</expression> 
<expression name="query_ME_link">
  select $link.get(1) 
  from TLink $link in $beliefbase.meansend 
  where $link.get(0).equals($component) 
  <parameter name="$component" class="String"/>
</expression> 
<expression name="query_contributions">
  select $link 
  from TContrib $link in $beliefbase.contributions 
  where $link.get(0).equals($component) 
  <parameter name="$component" class="String"/>
</expression> 
<expression name="query_dependencies">
  select $link 
  from TDependency $link in $beliefbase.dependencies 
  where $link.getWhyGoal().equals($component) 
  <parameter name="$component" class="String"/>
</expression> 
</expressions> 
<properties> 
<!--Only log outputs >= level are printed. -->
<!--The default parent handler prints out log messages on the console. -->
</properties> 
<initialstates>  
  <initialstate name="default"/>
</initialstates> 
</agent>
**Generated Plan files**

Following, some example generated Java plans are shown. The other plans needed for the example follow the same structure. The three standard plans, DispatchGoalPlan.java, GoalRequestPlan.java, and InformChangePlan.java are copied without change from the plan-skeletons library, shown at the end of Appendix C.

- `./t2x/generated/SearchActor/MetaPlan_find_word_description.java`
- `./t2x/generated/SearchActor/ANDGoalPlan_find_in_Internet.java`
- `./t2x/generated/SearchActor/MetaPlan_find_in_encyclopaedia.java`
- `./t2x/generated/SearchActor/RealPlan_Wikipedia.java`

```java
package t2x.generated.SearchActor;
import t2x.util.plans.AMetaPlan;

public class MetaPlan_find_word_description extends AMetaPlan {
    //Override
    //protected ICandidateInfo selection(ICandidateInfo[] applicables) {
    //    return applicables[0];
    //}
}
```

```java
package t2x.generated.SearchActor;
import t2x.util.plans.BaseANDDispatchGoalPlan;
public class ANDGoalPlan_find_in_Internet extends BaseANDDispatchGoalPlan {
    // for a custom implementation:
    // protected void goalsANDdispatch(String param) {...}
}
```

```java
package t2x.generated.SearchActor;
import t2x.util.plans.BaseANDDispatchGoalPlan;
public class ANDGoalPlan_find_in_Internet extends BaseANDDispatchGoalPlan {
    // for a custom implementation:
    // protected void goalsANDdispatch(String param) {...}
}
```
Appendix A: Example ADF XML file and plans

```java
/**
 * @author Mirko Morandini
 * Project JadexWork, file created on 4-mag-2006
 */
package t2x.generated.SearchActor;

import t2x.util.plans.AMetaPlan;

public class MetaPlan_find_in_encyclopaedia extends AMetaPlan {
    @Override
    protected ICandidateInfo selection(ICandidateInfo[] applicables) {
        return applicables[0];
    }
}

/*
 * ./t2x/generated/SearchActor/RealPlan_Wikipedia.java
 * 
 * @author Mirko Morandini
 * Project JadexWork, created on 3-mag-2006
 */
package t2x.generated.SearchActor;

import t2x.util.plans.BaseRealPlan;

public class RealPlan_Wikipedia extends BaseRealPlan {
    @Override
    public void body() {
        String param = (String) getParameter("param").getValue();
        //String resource = (String) useResource("RESOURCE");
        //****Business logic****

        String capabilityName=getPlanName();
        requestCapability(capabilityName, param);
        //********************
        getParameter("result").setValue(param);
        //produceResource("RESOURCE", result);
    }
}
```
Appendix B:
Plan and component library

All Java and XML file listings were generated using the following shell-skript and pretty-printed using the Eclipse Java- and XMLBuddy editors. All empty lines were removed.

```
find ./t2x/{producer,util} -name "*.java" -exec cat border_b.txt \;
 -exec echo {} \; -exec cat -s border_e.txt {} \; > all.java
```

## Components

./t2x/util/components/Components.java
./t2x/util/components/TComp.java
./t2x/util/components/TContrib.java
./t2x/util/components/TDependency.java
./t2x/util/components/TGoal.java
./t2x/util/components/TLink.java
./t2x/util/components/TSoftgoal.java

```java
/*
 * @author Mirko Morandini
 * Project Jadexwork, file created on 31-mag-2006
 */
package t2x.util.components;
import java.util.Hashtable;
/**
 * Class with only static methods, holds hashtables to store Tropos components (Goals and Softgoals).
 */
public class Components {
    private static Hashtable<String, TSoftgoal> softgoals = new Hashtable<String, TSoftgoal>();
    private static Hashtable<String, TGoal> goals = new Hashtable<String, TGoal>();
    public static TSoftgoal createSoftgoal(String name, double importance) {
        if (softgoals.get(name) == null) {
            softgoals.put(name, new TSoftgoal(name, importance));
        }
        return softgoals.get(name);
    }
    public static TGoal createGoal(String name, String decomp) {
        if (goals.get(name) == null) {
            goals.put(name, new TGoal(name, decomp));
        }
        return goals.get(name);
    }
}
```
Appendix B: Plan and component library

```java
public static TSoftgoal getSoftgoal(String name) {
    return softgoals.get(name);
}

public static TGoal getGoal(String name) {
    return goals.get(name);
}
```

```java
package t2x.util.components;

/**
 * A general Tropos component.
 */
public abstract class TComp {
    public String name;
    public TComp(String name) {
        this.name = name;
    }
    public abstract boolean isGoal();
    public abstract boolean isPlan();
    public abstract boolean isSoftgoal();
    public String getName() {
        return name;
    }
}
```

```java
package t2x.util.components;
import jadex.util.Tuple;

/**
 * A Tropos contribution link to a Softgoal.
 * Can define fixed values for the different contribution modes (++ to --)
 */
public class TContrib extends Tuple {
    public TContrib(String begin, String end, String contrib) {
        super(new Object[]{begin, end, contrib});
    }
    public TSoftgoal getSoftgoal() {
        return Components.getSoftgoal((String)get(1));
    }
    public String getContrib() {
        return (String)get(2);
    }
    public double getContribValue() {
        double w = 0;
        String cont = getContrib();
        if (cont.equals("++")) w = 0.8;
        else if (cont.equals("+")) w = 0.4;
        else if (cont.equals("-")) w = -0.4;
        else if (cont.equals("--")) w = -0.8;
        return w;
    }
    public double getContribRating() {
        double imp = getImportance();
        double cont = getContribValue();
        System.out.println("[TContrib]" + get(0) + " + get(1) + ": " + cont + ", w*imp = " + cont * imp);
        return cont * imp;
    }
}
```

```java
package t2x.util.components;

/**
 * @author Mirko Morandini
 * Project Jadexwork, file created on 31-mag-2006
 */
```
package t2x.util.components;
import jadex.util.Tuple;

/**
 * A Tropos contribution link to a Softgoal.
 * Can define fixed values for the different contribution modes (++, to --)
 */
public class TDependency extends Tuple {
    public TDependency(String why, String dependum, String dependee) {
        super(new Object[]{why, dependum, dependee});
    }
    public String getWhyGoal() {
        return (String) get(0);
    }
    public String getDependumGoal() {
        return (String) get(1);
    }
    public String getDependeeActor() {
        return (String) get(2);
    }
}

package t2x.util.components;
import javax.imag.Ordinance;
import java.util.ArrayList;
import java.util.List;
import t2x.util.plans.AMetaPlan;

/**
 * A Tropos goal component with methods to get recursively the contributions rating of a goal-decomposition tree.
 */
public class TGoal extends TComp {
    private String decomp; //"OR", "AND", "Means-end", "Delegation"
    public boolean isSoftgoal() {
        return false;
    }
    public boolean isGoal() {
        return true;
    }
    public boolean isPlan() {
        return false;
    }
    public TGoal(String name, String decomp) {
        super(name);
        this.decomp = decomp;
    }
    /**
     * Calculates recursively the contributions rating of a goal-decomposition tree.
     * @param caller The calling plan itself (usually "this").
     * @return a floating value (usually in [-1,1]) for the rating.
     */
    public double getRating(AMetaPlan caller) {
        //calc contribution of the direct softgoal contributions
        double rating = caller.calcSoftgoalContributions(this.name);
        System.out.println("I am in goal "+name+", decomp "+decomp+", contrib. here:");
        "+rating);
        if (decomp.equals("ME")) {
            List<String> plans = getAllDecompositionPlans(caller);
            double r = -1;
            for (String planame : plans) {
                r = Math.max(r, caller.calcSoftgoalContributions(planame));
            }
            rating += r;
        } else if (decomp.equals("OR")) {
            List<TGoal> goals = getAllDecompositionGoals(caller);
            double r = -1;
            for (TGoal goal : goals) {
                //recursively calc. rating
                r = Math.max(r, goal.getRating(caller));
            }
            rating += r;
        } else if (decomp.equals("AND")) {
            double r = 0;
            for (TGoal goal : goals) {
                r = goal.getRating(caller);
            }
        }
    }
}
Appendix B: Plan and component library

```java
{ 
    rating+=r;
    } //add delegation
    return rating;
}
/**
 * Gets all child plans from the belief base on Means-end.
 * @param caller The calling plan itself (usually "this").
 * @return The planame.
 */
@SuppressWarnings("unchecked")
public List<String> getAllDecompositionPlans(Plan caller){
    List execute = (List)caller.getExpression("query_ME_link").execute("$component", name);
    List <String>list = execute;
    return list;
}
/**
 * Gets all child goals from the belief base on AND/OR decomposition.
 * @param caller The calling plan itself (usually "this").
 * @return The planame.
 */
@SuppressWarnings("unchecked")
public List<TGoal> getAllDecompositionGoals(Plan caller){
    List execute = (List)caller.getExpression("query_link").execute("$component", name);
    List <TGoal>list = execute;
    return list;
}
/**
 * @return Returns the decomposition type ("OR", "AND", "Means-end" or "Delegation").
 */
public String getDecomp() {
    return decomp;
}
/**
 * @return Returns the name.
 */
public String getName() {
    return name;
}
}/**
.//t2x/util/components/TLink.java
*****************************************************************************/

package t2x.util.components;
import jadex.util.Tuple;

public class TLink extends Tuple {
    public TLink(String begin, String end) {
        super(begin, end);
    }
}/**
 */
public TLink(String begin, String end, int priority) {
    super(new Object[]{begin, end, priority}); // used java 1.5 Integer boxing!
}

public TGoal getGoal(int idx) {
    return Components.getGoal((String) get(idx));
}/**
 * Gets the priority, if it was set in the constructor.
 * @return the priority if set, 0 otherwise.
 */
public Integer getPriority() {
    return size() > 2 ? (Integer) get(2) : 0;
}*/
package t2x.util.components;
/**
 * @author Mirko Morandini
 * Project JadexWork, file created on 23-mag-2006
 */
public class TSoftgoal extends TComp {
    private double imp;
    public TSoftgoal(String name, double importance){
        super(name);
        imp=importance;
    }
    public boolean isSoftgoal() {return true;}
    public boolean isGoal() {return false;}
    public boolean isPlan() {return false;}
    public double getImportance() {
        return imp;
    }
    public void setImportance(double imp) {
        this.imp=imp;
    }
}*/

Jadex Plan extensions

./t2x/util/plans/AMetaPlan.java
./t2x/util/plans/APlan.java
./t2x/util/plans/BaseANDDispatchGoalPlan.java
./t2x/util/plans/BaseDispatchGoalPlan.java
./t2x/util/plans/BaseGoalRequestPlan.java
./t2x/util/plans/BaseInformChangePlan.java
./t2x/util/plans/BaseRealPlan.java

/*********************/
/**
 * @author Mirko Morandini
 * Project JadexWork, file created on 4-mag-2006
 */
package t2x.util.plans;
import java.util.List;
import t2x.util.components.Components;
import t2x.util.components.TContrib;
import t2x.util.components.TGoal;
import jadex.runtime.ICandidateInfo;
import jadex.runtime.Plan;

/*
 * An abstract metaplan to use for the Jadex metaplans.
 * Contains the selection intelligence, that can also be overridden.
 */
public abstract class AMetaPlan extends Plan {
    /*
    *
    *
    public AMetaPlan() {
        super();
    }
    @Override
    public void body() {
        ICandidateInfo[] apps = (ICandidateInfo[])getParameterSet("applicables").getValues();
        ICandidateInfo sel = null;
        getLogger().info("MetaPlan "+this+" started, plan "+sel.getPlan(this).getName()+" dispatched.");
        getParameterSet("result").addValue(sel);
    }
    // example selection method
    //
    protected abstract ICandidateInfo selection(ICandidateInfo[] applicables);
    //
    
    protected ICandidateInfo selection(ICandidateInfo[] applicables) {
        double r, max=-100;
        ICandidateInfo best = null;
        for(ICandidateInfo app:applicables){
            r=rate(app);//takes the plan name
            if (r>max){
                max=r;
                best=app;
            }
        }
        System.out.println("Applicable plan: "+planame);
        if (planame.startsWith(APlan.PLANPREFIX)) {
            //MEANS-END
            String plan=planame.substring(APlan.PLANPREFIX.length());
            rating = calcSoftgoalContributions(plan);
        }
        System.out.println("rating: "+r);
        System.out.println("best plan: "+best);
        return best;
    }
    /*
    * Calculates the softgoal contribution recursively for the whole hierarchy,
    * starting from this plan. Relies on the beliefbase definitions to retrieve the goal decompositions.
    * @return A value in [0..1] that expresses the softgoal rating at this level of the hierarchy.
    */
    public double rate(ICandidateInfo app) {
        double rating=0;
        String planame=app.getPlan(this).getModelElement().getName(); //takes the plan name
        System.out.println("Applicable plan: "+planame);
        if (planame.startsWith(APlan.PLANPREFIX)) {
            //MEANS-END
            String plan=planame.substring(APlan.PLANPREFIX.length());
            rating = calcSoftgoalContributions(plan);
        } else if (planame.startsWith(APlan.GOALPREFIX)) {
            //OR-DECOMPOSITION
            String goalname=planame.substring(APlan.GOALPREFIX.length());
            if (there can be no AND-plan here! (AND-plans can be found only after AND-goals!)
            String goalname=planame.substring(APlan.GOALPREFIX.length());
            return goalname.length();
        } else if (planame.startsWith(APlan.PLANPREFIX)) {
            //can retrieve the goal directly, without an expression on beliefbase!
            TGoal goal=Components.getGoal(goalname);
            assert goal!=null;
    */
//returns the goal's rating: recursive rating calculation down to the leaf
rating=goal.getRating(this);
} else
System.out.println("WARNING: Jadex Plan Type unknown: "+planame!");
return rating;
/**
* Calculates the softgoal contribution for a plan in means-end or for a goal (uses the dispatch-plans).
* @param name
* @return
*/
@SuppressWarnings("unchecked")
public double calcSoftgoalContributions(String name) {
  double contrib=0;
  //Write expression directly here!
  List<TContrib> clist = (List<TContrib>)getExpression("query_contributions").execute("$component", name);
  System.out.println("Contribution list for plan "+name+": "+clist);
  for (TContrib c:clist){
    contrib+=c.getContribRating();
  }
  return contrib;
}
if (reply.getParameter("performative").getValue().equals(SFipa.FAILURE)) {
    requestFAILURE(task, actor, param, monitorTask);
} else if (reply.getParameter("performative").getValue().equals(SFipa.INFORM)) {
    requestINFORM(task, actor, param, monitorTask);
} else if (reply.getParameter("performative").getValue().equals(SFipa.REFUSE)) {
    requestREFUSE(task, actor, param, monitorTask);
} else if (reply.getParameter("performative").getValue().equals(SFipa.NOT_UNDERSTOOD)) {
    requestN_U(task, actor, param, monitorTask);
}

monitor=null;

protected void requestAGREE(String task, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " gave AGREE on " + monitorTask+" with " + task + ";
    getLogger().info(cont);
}

protected void requestINFORM(String task, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " INFORMS that " + monitorTask + " with " + task + " was successfuly performed."
    getLogger().info(cont);
}

protected void requestFAILURE(String task, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " informs that " + monitorTask + " with " + task + "+ parameters " + param + " has FAILED!";
    getLogger().warning(cont);
    fail();
}

protected void requestN_U(String task, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " informs that " + monitorTask + " with " + task + "+ parameters " + param + " was not understood!";
    getLogger().warning(cont);
    fail();
}

protected void requestREFUSE(String task, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " informs that " + monitorTask + " with " + task + "+ parameters " + param + " was refused!
    getLogger().warning(cont);
    fail();
}

public void failed() {
    Exception e = getException();
    if (e.toString().equals("jadex.runtime.TimeoutException")) {
        getLogger().warning(" сайт");
    } else {
        getLogger().warning(e + ". Problem: "+reason+".");
    }
}
public abstract class BaseANDDispatchGoalPlan extends APlan {
  public BaseANDDispatchGoalPlan() {
    super();
    getLogger().info("Created: " + this + ".");
  }

  public void body() {
    String param = (String) getParameter("param").getValue();
    // the new prototype uses resources and produce/use dependencies to propagate results, if
    // wanted by the designer.
    // goalDependencies(param);
    goalsANDdispatch(param);
  }

  @SuppressWarnings("deprecation")
  protected void goalDependencies(String param) {
    // get the triggering goal name
    String goalname = getRootGoal().getModelElement().getName();
    // get the goal object created in the beliefbase
    TGoal g = Components.getGoal(goalname);
    if (g == null) {
      getLogger().warning("Goal " + goalname + " not found in Belief Base!");
      fail(); // propagates the failure directly to the parent goal.
    }
    // get all decomposition goals for this goal from the beliefbase
    List<TDependency> dependencies = g.getAllDependencies(this);
    // dispatch all dependencies in AND (one fails->all fails)
    for (TDependency dep : dependencies) {
      request(dep.getDependumGoal(), dep.getDependeeActor(), param, "resolve dependency", "dependency_request");
    }
  }

  @Override
  protected void requestFAILURE(String goal, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " informs that dependent goal " + goal + " with parameters " + param + " has FAILED!";
    getLogger().warning(cont);
    fail();
  }

  @Override
  protected void requestINFORM(String goal, String actor, String param, String monitorTask) {
    String cont = "Actor " + actor + " informs that dependent goal " + goal + " with parameters " + param + " was reached."
    getLogger().info(cont);
  }

  * Overwriteable! Attention: used deprecated method "Plan.getRootGoal"
  * @param param
  * @Override
  @SuppressWarnings("deprecation")
  protected void goalsANDdispatch(String param) {
    Object result = param;
    // get the triggering goal name
    String goalname = getRootGoal().getModelElement().getName();
    // get the goal object created in the beliefbase
    TGoal g = Components.getGoal(goalname);
    if (g == null) {
      getLogger().warning("AND goal failure: No goal decomposition for " + goalname + " found in Belief Base!");
      fail(); // propagates the failure directly to the parent goal.
    }
    // get all decomposition goals for this goal from the beliefbase
    List<TGoal> goals = g.getAllDecompositionGoals(this);
    if (goals.size() < 1) {
      fail();
      // dispatch all goals in AND (one fails->all fails)
      for (TGoal tGoal : goals) {
        monitor=new Object[] {"dispatch subgoal", g.getName(), tGoal.getName()};
        IGoal goal = createGoal(tGoal.getName());
        goal.getParameter("param").setValue(param);
        try {
          dispatchSubgoalAndWait(goal, TIMEOUT);
          result = goal.getParameter("result").getValue();
        } catch (GoalFailureException e) {
          ...
Appendix B: Plan and component library

```java
getLogger().warning("Local goal failure (goal " + tGoal.name + ", content: " + param + ">");
} fail();// propagates the first failure directly to the parent goal (AND!).

monitor=null;
// return the results from the subgoal to the triggering goal
// only the last result is returned, all others are discarded by default
getParameter("result").setValue(result);// sets result in the triggering goal
}

@SuppressWarnings("serial")
public abstract class BaseDispatchGoalPlan extends APlan {
  public String requestedgoal;
  public BaseDispatchGoalPlan(final String requestedGoal) {
    super();
    this.requestedgoal=requestedGoal;
    getLogger().info("Created: "+this+" with goal "+this.requestedgoal+".");
  }

  public void body() {
    String param="";
    param=(String)getParameter("param").getValue();
    IGoal goal=createGoal(requestedgoal);
    goal.getParameter("param").setValue(param);
    try{
      dispatchSubgoalAndWait(goal, TIMEOUT);
      Object result=goal.getParameter("result").getValue();
      //return the results from the subgoal to the triggering goal
      getParameter("result").setValue(result);
    }catch(GoalFailureException e){
      getLogger().warning("Local goal failure (goal "+requestedgoal+", content: "+param+");
    fail();//propagates the failure to the parent goal.
  }
}
}
```

```java
package t2x.util.plans;
import jadex.runtime.GoalFailureException;
import jadex.runtime.IMessageEvent;
import java.util.StringTokenizer;

public abstract class BaseGoalRequestPlan extends APlan {
  public String requestedgoal;
  public BaseGoalRequestPlan(final String requestedGoal) {
    super();
    this.requestedgoal=requestedGoal;
    getLogger().info("Created: "+this+" with goal "+this.requestedgoal+".");
  }

  public void body() {
    String request=(String)((IMessageEvent)getInitialEvent()).getContent();
    //if(event.getEventclass().equals("eventclass_message")){
    StringTokenizer stok=new StringTokenizer(request," ");
    int cnttokens=stok.countTokens();
    if(cnttokens>=2){
      //
      getInitialEvent().setMessageContent(" ");
      getParameter("result").setValue(result);
    }
    }
}
```

```java
package t2x.util.plans;
import jadex.runtime.GoalFailureException;
import jadex.runtime.IMessageEvent;
import java.util.StringTokenizer;

public abstract class BaseGoalRequestPlan extends APlan {
  public String requestedgoal;
  public BaseGoalRequestPlan(final String requestedGoal) {
    super();
    this.requestedgoal=requestedGoal;
    getLogger().info("Created: "+this+" with goal "+this.requestedgoal+".");
  }

  public void body() {
    String request=(String)((IMessageEvent)getInitialEvent()).getContent();
    //if(event.getEventclass().equals("eventclass_message")){
    StringTokenizer stok=new StringTokenizer(request," ");
    int cnttokens=stok.countTokens();
    if(cnttokens>=2){
    
```
@SuppressWarnings("unused")
String action = stok.nextToken(); // the goal name
String param = stok.nextToken();
IGoal goal = createGoal(requestedgoal);
goal.getParameter("param").setValue(param);
goalAgree(goal.getName(), param);
try {
    dispatchSubgoalAndWait(goal);
    Object result = goal.getParameter("result").getValue();
goalSuccess(goal.getName(), param, result);
} catch (GoalFailureException e) {
    getLogger().warning("Goal " + goal.getName() + " failed (content: "+param+ ").");
goalFailure(goal.getName(), param);
}
else {
    getLogger().warning("Request format not correct: "+request);
    requestFailure(request);
}
/**
 * @param name
 * @param param
 * @param result
 */
protected abstract void goalSuccess(String name, String param, Object result);
/**
 * @param name
 * @param param
 */
protected abstract void goalFailure(String name, String param);
/**
 * @param request
 */
protected abstract void requestFailure(String request);
/**
 * @param name
 * @param param
 */
protected abstract void goalAgree(String name, String param);

package t2x.util.plans;
import jadex.runtime.IMessageEvent;
import java.util.StringTokenizer;
import t2x.util.components.Components;
import t2x.util.components.TSoftgoal;
public abstract class BaseInformChangePlan extends APlan {
    String request = ""
    public BaseInformChangePlan() {
        super();
        getlogger().info("Created: " + this + ".");
    }
    public void body() {
        request = (String) ((IMessageEvent) getInitialEvent()).getContent();
        TSoftgoal sg;
        if (request.getClass().equals("eventclass_message")) {
            StringTokenizer stok = new StringTokenizer(request, " ");
            int cnttokens = stok.countTokens();
            if (cnttokens >= 3) {
                @SuppressWarnings("unused")
                String action = stok.nextToken(); // "change"
                String component = stok.nextToken(); // the softgoal/resource name
                String content = stok.nextToken(); // the new value
                if ((sg = Components.getSoftgoal(component)) != null) {
                    try {
                        double imp = Double.valueOf(content);
                        sg.setImportance(imp);
                    } catch (NumberFormatException e) {
                        getLogger().warning("Content not a double number.");
                        fail();
                    }
                }
            }
        }
    }
Appendix B: Plan and component library

```java
} else if (getBeliefbase().containsBelief(RESOURCEPREFIX + component)) {
    try {
        Class c = getBeliefbase().getBelief(RESOURCEPREFIX + component).getClazz();
        Object o = c.cast(content);
        getBeliefbase().getBelief(RESOURCEPREFIX + component).setFact(o);
    } catch (RuntimeException e) {
        getLogger().warning("Error storing the content, maybe due to a casting problem.");
        fail();
    }
} else {
    getLogger().warning("Component ", component + " not found either in the belief base nor in the softgoals base.");
    fail();
} else {
    getLogger().warning("Inform change format not correct: ", request + ". Expected: " + request + " [softgoal|resource] value\n".");
    requestFailure(request);
}
/**
 * @param request
 */
protected void requestFailure(String cont) {
    getLogger().warning(cont);
    sendMessage(((IMessageEvent) getInitialEvent()).createReply("failure", cont));
}
/**
 * @package t2x.util.plans
 */
public abstract class BaseRealPlan extends APlan {
    protected String capabilitiesAgent = "CapabilitiesAgent";
    public BaseRealPlan() {
        super();
        getLogger().info("RealPlan ", getName() + " started. Param: " + getParameter("param").getValue());
    }
    protected Object useResource(String name) {
        Object res = null;
        try {
            res = getBeliefbase().getBelief(RESOURCEPREFIX + name).getFact();
        } catch (RuntimeException e) {
            getLogger().warning("UseResource: Resource ", name + " not found!");
        }
        return res;
    }
    protected void produceResource(String name, Object o) {
        try {
            getBeliefbase().getBelief(RESOURCEPREFIX + name).setFact(o);
        } catch (RuntimeException e) {
            getLogger().warning("ProduceResource: Resource ", name + " not found!");
        }
    }
    protected void requestCapability(String capability, String param) {
        request(capability, capabilitiesAgent, param, "request Capability", "capability_request");
    }
    @Override
    protected void requestINFORM(String capability, String actor, String param, String monitorTask) {
        String cont = "Actor " + capabilitiesAgent + " informs that capability " + capability + " with parameters " + param + " was correctly executed."
        ....
    }
    //
    /t2x/util/plans/BaseRealPlan.java
    /*******************************************************************************/
    /**
     * @author Mirko Morandini
     * Project JadxWork, file created on 29-mag-2006
     **/
     package t2x.util.plans;
    */
```
```java
private String capabilitiesAgent;

@Override
protected void requestFAILURE(String capability, String actor, String param, String monitorTask) {
    String cont = "Actor " + capabilitiesAgent + " informs that capability " + capability + " with parameters " + param + " has FAILED!";
    getLogger().warning(cont);
    fail();
}

@Override
protected void requestAGREE(String capability, String actor, String param, String monitorTask) {
    getLogger().info("AGREE form " + capabilitiesAgent + " for capability " + capability + ");
}

public String getPlanName() {
    return getRPlan().getModelElement().getName().substring("RealPlan_".length());
}
```
Appendix C: The Code Generation Tool t2x

Class diagrams

Class diagram with the main class `Producer`, the `AgentDefinition` class containing all information extracted from TROPOS goal decomposition diagrams necessary for the code generation, and the `AgentWriter` class, responsible for producing the ADF file and writing the Java Plan classes.
Class diagram for the classes used in the AgentDefinition class to store all relevant data for the concepts of softgoal, goal and plan, in a tree-like structure (correctly, a DAG) that resembles the TROPOS goal diagram.

**Java code listing**

./t2x/producer/Producer.java
./t2x/producer/AgentDefinition.java
./t2x/producer/AgentWriter.java
./t2x/producer/Const.java
./t2x/producer/ElementContainer.java
./t2x/producer/GoalContainer.java
./t2x/producer/PlanContainer.java
./t2x/producer/SoftgoalContainer.java
/**
 * This tool produces a BDI-based implementation in Jadex from a Tropos Actor diagram modelled in TAOM.
 * Run it in Eclipse in a configuration with the following arguments:
 * "${resource_loc}" ${string_prompt}
 * Select the Tropos Model, Run the configuration, input the actor name in the prompt.
 * @param args
 */

public static void main(String args[]) {
    String filename = "model/BigExample.tropos"; // take from args in final version
    String actorname = null;
    if (args==null || args.length==0) {
        System.out.println("No input arguments found. \n" + "Call me with [ModelFilename.troopos] [Actor_Name]\n" + "Now I take the default model and the first actor.\n");
    } else {
        if (args.length>0) {
            filename=args[0];
        }
        if (args.length>1) {
            actorname=args[1];
        }
    }
    // load a TAOM-generated TROPOS Model from an XML file
    tn = new TroposNavigator(filename);
    print("TROPOS file " + filename + " loaded.\n");
    List<Actor> actors = tn.getActors();
    tn.printContents("Available actors: \n", actors);
    for (Actor a : actors) {
        print("-------------------------------------\n");
        tn.printContents("All hard goals for " + a.getName(), tn.getHardGoals(a));
        tn.printContents("Root     goals for " + a.getName(), tn.getRootGoals(a));
        tn.printContents("All soft goals for " + a.getName(), tn.getSoftGoals(a));
        tn.printContents("All   plans for " + a.getName(), tn.getPlans(a));
        tn.printContents("Means plans for " + a.getName(), tn.getAllMeansEndPlans(a));
    }
    print("**************************************************************\n");
    Actor a=null;
    if (actorname!=null) {
        print("I try to take Actor " + actorname);
        a=tn.getActor(actorname);
    }
    if (a==null) {
        a = actors.get(0);
        print("I take Actor " + a.getName());
    }
    AgentDefinition ad = new AgentDefinition(a);
    // analyse all root goals
    for (HardGoal rootgoal : tn.getRootGoals(a)) {
        print(" Now I analyse root goal: " + rootgoal.getName());
        Const type, request;
        if (tn.isDelegated(rootgoal)) {
            type = Const.ACHIEVE;
            request=Const.REQUEST;
        } else {
            type=Const.MAINTAIN;
            type = Const.ACHIEVE; // For now only achieve is implemented, this has to be
        request=Const.NONE;
        // maintain-goal!"
Appendix C: The Code Generation Tool t2x

```java
GoalContainer gc = ad.createGoal(rootgoal, type);
    gc.setRequest(request);
    ad.addRootGoal(gc);
    addGoal(rootgoal, gc, ad);

    // write the agent (xml+java plan bodies) to the output directory
    ad.writeAgent();
}

private static void addGoal(Goal g, GoalContainer gc, AgentDefinition ad) {
    addContributions(g, gc, ad);
    if (tn.isDelegated(g))
        gc.setRequest(Const.REQUEST);
    if (tn.isBooleanDecAND(g)) {
        List<Goal> declist = tn.getBooleanDec(g);
        // sets decomposition flag and creates the AND-Plan (call only one time!)
        gc.createDecomposition(Const.AND);
        for (Goal dec : declist) {
            // addDecomp adds the new goal to container and goalbase and, if needed (OR, M-E)
            // organizes dispatch goals
            GoalContainer decont = ad.createGoal(dec, Const.ACHIEVE);
            gc.addDecomp(decont);
            addGoal(dec, decont, ad);
        }
    } else if (tn.isBooleanDecOR(g)) {
        List<Goal> declist = tn.getBooleanDec(g);
        // sets decomposition flag and creates the Metagoal+plan (call only one time!)
        gc.createDecomposition(Const.OR);
        for (Goal dec : declist) {
            // addDecomp adds the new goal and, if needed (OR, M-E) organizes dispatch goals
            GoalContainer decont = ad.createGoal(dec, Const.ACHIEVE);
            gc.addDecomp(decont);
            addGoal(dec, decont, ad);
        }
    }
    if (tn.isMeansEndDec(g)) {
        List<Plan> melist = tn.getMeansEndMeanPlans(g);
        // sets decomposition flag and creates the Metagoal+plan,
        // shall be the same than with OR! They could also be mixed in this
        // implementation!
        gc.createDecomposition(Const.ME);
        for (Plan p : melist) {
            PlanContainer pc = ad.createPlan(p);
            gc.addMERealPlan(pc);
            addPlan(p, pc, ad);
        }
    }
    if (tn.isGoalWhyDependency(g)) {
        Dependency dep = tn.getGoalDependencies(g) {
            String goal = AgentDefinition.fill(tn.getDependumGoalFromDependency(dep).getName());
            String actor = AgentDefinition.fill(tn.getActorFromDependency(dep).getName());
            gc.addDependency(goal, actor);
        }
    }

    private static void addContributions(TroposModelElement m, ElementContainer ec, AgentDefinition ad) {
        if (tn.hasContributions(m)) {
            System.out.println("All Contributions of " + m.getName() + ": ");
            for (FContribution c : tn.getContributions(m)) {
                System.out.println(c.getSource().getName() + " | " + c.getMetric() + " | " + c.getTarget().getName());
                if (c.getTarget() instanceof FSoftGoal) {
                    FSoftGoal sg = (FSoftGoal) c.getTarget();
                    SoftgoalContainer sgcont = ad.createSoftgoal(sg);
                    ec.addContribution(sgcont, c.getMetric());
                }
            }
        }
    }
```

126
private static void addPlan(Plan p, PlanContainer pc, AgentDefinition ad) {
    addContributions(p, pc, ad);
    // RESOURCES (USE/PRODUCE) TO BE IMPLEMENTED, HERE AND IN THE NAVIGATOR!!
}

public static void print(String s) {
    System.out.println(s);
}

package t2x.producer;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.List;
import it.itc.sra.taom4e.model.core.informalcore.*;
import it.itc.sra.taom4e.model.core.informalcore.formalcore.FSoftGoal;
public class AgentDefinition {
    private String agentname;
    private LinkedList<GoalContainer> rootlist = new LinkedList<GoalContainer>();
    public Hashtable<String, SoftgoalContainer> softgoalbase;
    public Hashtable<String, GoalContainer> goalbase;
    public Hashtable<String, PlanContainer> planbase;
    public AgentDefinition(Actor a) { 
        goalbase = new Hashtable<String, GoalContainer>();
        softgoalbase = new Hashtable<String, SoftgoalContainer>();
        planbase = new Hashtable<String, PlanContainer>();
        agentname = fill(a.getName());
    }
    public void addRootGoal(GoalContainer rootgoal) {
        rootlist.add(rootgoal);
    }
    public List<GoalContainer> getRootGoalList() {
        return rootlist;
    }
    /**
     * @param sg
     * @return
     */
    public SoftgoalContainer createSoftgoal(FSoftGoal sg) {
        SoftgoalContainer gc = new SoftgoalContainer(sg); // needed here to get surely the right name
        if (softgoalbase.containsKey(gc.name))
            return softgoalbase.get(gc.name); // just created sg disposed!
        softgoalbase.put(gc.name, gc);
        return gc;
    }
    public GoalContainer createGoal(Goal goal, Const type) {
        GoalContainer gc = new GoalContainer(goal, type);
        if (goalbase.containsKey(gc.name))
            return goalbase.get(gc.name);
        goalbase.put(gc.name, gc);
        return gc;
    }
    public PlanContainer createPlan(Plan p) {
        PlanContainer gc = new PlanContainer(p);
        if (planbase.containsKey(gc.name))
            return planbase.get(gc.name);
        planbase.put(gc.name, gc);
        return gc;
    }
    /**
     * Fills all spaces in a string with underscores '_'.
     * @param name
     * @return
     */
    public static String fill(String s) {
        // Code to fill string with underscores
    }
}
return s.replace(' ','_');
/**
 * @return
 */
public void writeAgent()
{
    new AgentWriter(this).writeAgent();
}
/** @return */
public String getAgentName()
{
    return agentname;
}
/**
 * @author Mirko Morandini
 * Project JadexWork, file created on 12-giu-2006
 * ITC-irst, Università di Trento, 2006.
 */
package t2x.producer;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.channels.FileChannel;
/**
 * Writes the agent from the internal representation
 */
public class AgentWriter
{
    private final String sourcepath = "src/t2x/producer/planskeletons/";
    private String path;
    private String packagename;
    // Strings that contain the parts of the ADF skeleton, read from file
    string header, body, footer;
    // Strings filled with content, to replace the placeholders in the adf skeleton.
    String bbdecomp = "", bbmeansend = "", bbcontrib = "", bbdepend="";
    String adfgoals = "", adfmetagoals = "", adfrequestplans = "",
    adfdispatchplans = "", adfandplans = "", adfrealtasks = "",
    adfrealplans = "", adfmetaplans = "", adfcalls = "",
    bbgoals = "", bbsoftgoals = "",
    //the output ADF XML file
    PrintWriter adfFile;
    //Has all the informations about the agent.
    AgentDefinition ad;
    /**
     * @param ad
     */
    public AgentWriter(AgentDefinition ad)
    {
        this.ad = ad;
        // path and packagename to set here manually for now!
        //path = "./JadexGenerated/src/t2x/generated/" + ad.getAgentName() + "/";
        path = "src/t2x/generated/" + ad.getAgentName() + "/";
        packagename = "t2x/generated" + ad.getAgentName();
        header = readFilesAsString(sourcepath + "agentheader.xml");
        body = readFilesAsString(sourcepath + "agentbody.xml");
        footer = readFilesAsString(sourcepath + "agentfooter.xml");
        File dir = new File(path);
        if (!dir.exists())
            if (!dir.mkdirs())
                System.err.println("Error: Can't create output directory " + dir + "\/");
                System.exit(1);
        try
        {
            String adf = ad.getAgentName() + ".agent.xml"
            adFFile = new PrintWriter(new BufferedWriter(new FileWriter(path + adf)));
        } catch (IOException e)
        {
            System.err.println("Error: Can't create output adf file.");
            e.printStackTrace();
        }
}
private
/**
 * Writes the whole Agent (ADF + Java plan bodies).
 */
public void writeAgent() {
    header = header.replace("$NAME", ad.getAgentName());
    header = header.replace("$PACKAGE", packagename);
    writeBBsSoftgoals();
    writeGoals();
    writePlans();
    writeDefaultJavaFiles();
    printADF();
    writeStartingFiles();
}
/**
 * Writes the batch files (windows) to start the platform and the agent.
 */
private void writeStartingFiles() {
    // Try to copy file with the right name.
    try {
        copyFile(sourcepath + "platform.bat", path + "platform.bat");
    } catch (IOException e) {
        System.err.println("Warning: platform start batch file not copied correctly.");
    }
}
/**
 * Copies to the output directory all the files where only the package-name changes.
 */
private void writeDefaultJavaFiles() {
    // only the internal package name changes
    dispatchGoalPlan = readFileAsString(sourcepath + "DispatchGoalPlan.java");
    dispatchGoalPlan.replace("$PACKAGE", packagename);
    writeFile(dispatchGoalPlan, path + "DispatchGoalPlan.java");
    goalRequestPlan = readFileAsString(sourcepath + "GoalRequestPlan.java");
    goalRequestPlan.replace("$PACKAGE", packagename);
    writeFile(goalRequestPlan, path + "GoalRequestPlan.java");
    informChangePlan = readFileAsString(sourcepath + "InformChangePlan.java");
    informChangePlan.replace("$PACKAGE", packagename);
    writeFile(informChangePlan, path + "InformChangePlan.java");
    // the other files are written directly in writePlans/writeGoals
}
/**
 * replaces all placeholders in the ADF skeleton and writes the ADF file.
 */
private void printADF() {
    header = header.replace("$BBGOALS", bbgoals);
    header = header.replace("$BBSOFTGOALS", bbssoftgoals);
    header = header.replace("$DECOMP", bbddecomp);
    header = header.replace("$MEANSEND", bbdmeansend);
    header = header.replace("$CONTRIB", bbcontrib);
    header = header.replace("$DEPENDENCIES", bbddepend);
    body = body.replace("$GOALS", adgoals);
    body = body.replace("$METAGOALS", admetagoals);
    body = body.replace("$REQUESTPLANS", adfdrequestplans);
    body = body.replace("$DISPATCHPLANS", adfdispatchplans);
    body = body.replace("$METAPLANS", adfmetaplans);
    body = body.replace("$REALPLANS", adfreaplans);
    footer = footer.replace("$EVENTS", adfevents);
    adffile.println(header);
    adffile.println(body);
    adffile.println(footer);
    adffile.close();
    Producer.print("ADF XML-file written to " + path + ".");
}
/**
 * Reads all softgoals from the softgoals list and writes them into the belief base.
 */
private void writeBBsSoftgoals() {
}
Appendix C: The Code Generation Tool t2x

```java
String pattern = "\t<\t<fact>Components.createSoftgoal("”$NAME”", $VALUE</fact>"n",
for (SoftgoalContainer goal : ad.softgoalbase.values()) {
    String fact = pattern.replace("$NAME", goal.getName());
    fact = fact.replace("$VALUE", Double.toString(goal.getImportance()));
    bbsoftgoals = bbsoftgoals.concat(fact);
}
}/**
 * writes all goals to the ADF file (to beliefbase, goals and plans section) and
 * organizes * (copies) the plan bodies. Works not recursive on the goal structure, but processes
 * all goals * in the list in sequence.
 */
private void writeGoals() {
    String bbpattern = "\t<\t<fact>Components.createGoal("”$NAME”", "$DECTYPE”</fact>"n";  
    String goalpattern = readFileAsString(sourcepath + "pattern_goal.xml");
    String metagolapattern = readFileAsString(sourcepath + "pattern_metagol.xml");
    String dispatchplanpattern = readFileAsString(sourcepath + "pattern_dispatchplan.xml");
    String dispatchANDPattern = readFileAsString(sourcepath + "pattern_dispatchANDplan.xml");
    String requestplanpattern = readFileAsString(sourcepath + "pattern_requestplan.xml");
    String eventpattern = readFileAsString(sourcepath + "pattern_event.xml");
    // from this files multiple files associated to goal/plannames have to be written
    String file_MetaPlan = readFileAsString(sourcepath + "MetaPlan_TMP.java");
    String file_ANDGoalPlan = readFileAsString(sourcepath + "ANDGoalPlan_TMP.java");
    for (GoalContainer goal : ad.goalbase.values()) {
        // write the goal to the beliefbase
        String name = goal.getName();  
        String fact = bpattern.replace("$NAME", name);
        fact = fact.replace("$DECTYPE", goal.getDecomposition().toString());
        bbgoals = bbgoals.concat(fact);
        String adfgoal = goalplanpattern.replace("$NAME", name);
        adfgoals = adfgoals.concat(adfgoal);
        if (goal.request == Const.REQUEST) {
            writeRequestplan(goal, requestplanpattern);
            writeEvent(goal, eventpattern);
        } else if (Const.NONE, no plan is written.
        if (goal.getParentGoals().size() > 0) {
            // write the dispatch plans and add a trigger for every parent goal
            writeRealplans(goal, dispatchplanpattern);
        } if (goal.getDecomposition() == Const.OR) {
            writeMetaGoalPlan(goal, metagoalpattern, metaplanpattern, file_MetaPlan);
            writeDispatchGoals(goal, dispatchplanpattern);(now made backwards from the
            child goal)
        } if (goal.getDecomposition() == Const.ME) {
            writeMetaGoalPlan(goal, metagoalpattern, metaplanpattern, file_MetaPlan);
            file_MetaPlan = readFileAsString(sourcepath + "MetaPlan_TMP.java");
            writeDispatchANDplan(goal, dispatchANDpattern, file_ANDGoalPlan);
        } // retrieve softgoal contributions
        for (SoftgoalContainer sg : goal.getContributions().keySet()) {
            addBBContrib(goal, sg, goal.getContributions().get(sg));
        } // create dependencies with other agents
        for (String[] dep : goal.getDependencies()) {
            String depende = StringDEPENDENCY(goal, depende);
            String dependee = StringDEPENDENCY(goal, dependee, "My" + dependee);
        }
    }
    private void writeMetaGoalPlan(GoalContainer g, String gpattern, String ppattern,
            String file_MetaPlan) {
        String adfgoal = gpattern.replace("$NAME", g.getName());
        adfmetagoals = adfmetagoals.concat(adfgoal);
        String adfplan = ppattern.replace("$NAME", g.getName());
        adfmetaplanpattern = adfmetaplanpattern.concat(adfplan);
    // copy and rename the plan body
    String file = file_MetaPlan.replace("$PACKAGE", packagename);
    file = file.replace(".java", g.getName());
    writefile(file, path + "MetaPlan_TMP.java");
```
private void writeRequestplan(GoalContainer g, String pattern) {
    String adfplan = pattern.replace("$NAME", g.getName());
    adfrequestplans = adfrequestplans.concat(adfplan);
}

private void writeEvent(GoalContainer g, String pattern) {
    String adfevent = pattern.replace("$NAME", g.getName());
    adfevents = adfevents.concat(adfevent);
}

/**
 * Writes the dispatch plans (with bodies) for every child goal
 *
 * @param goal The goal.
 * @param pattern
 */

private void writeDecompositionLinks(GoalContainer goal, String pattern) {
    String params="", results="", triggers="";
    for (GoalContainer parent : goal.getParentGoals()) {
        params=params+params.concat("<goalmapping ref=""+parent.getName()+""/><param name=""+parent.getName()+"" result=""/"">\n"));
        triggers=triggers.concat("<goal ref=""+parent.getName()+"" trigger=""/"">\n"));
        addBBDecomp(parent, goal);
    }
    String adfplan = pattern.replace("$PARAMS", params);
    adfplan = adfplan.replace("$RESULTS", results);
    adfplan = adfplan.replace("$TRIGGERS", triggers);
    adfplan = adfplan.replace("$CHILDS", goal.getName());
    adddispatchplans = adddispatchplans.concat(adfplan);
}

private void writeDispatchANDPlan(GoalContainer g, String pattern, String file_ANDGoalPlan) {
    String adfplan = pattern.replace("$NAME", g.getName());
adfandplans = adfandplans.concat(adfplan);
    for (GoalContainer child : g.getDecompGoals()) {
        addBBDecomp(g, child, g.getPriority() /);
    }
    // copy and rename the plan body
    String file = file_ANDGoalPlan.replace("$PACKAGE", packagename);
    file = file.replace("TMP", g.getName());
    writeToFile(file, path + "ANDGoalPlan" + g.getName() + "_.java");
}

public void addBBDecomp(ElementContainer src, ElementContainer dest) {
    String pattern = "<t\t<fact>new TLink("$SRC", "$DEST")</fact>
";
    String fact = pattern.replace("$SRC", src.getName());
    fact = fact.replace("$DEST", dest.getName());
    bbdealloc = bbdealloc.concat(fact);
}

public void addBBMeansEnd(ElementContainer src, ElementContainer dest, int priority) {
    String pattern = "<t\t<fact>new TMeansEnd("$SRC", "$DEST", $PRIORITY</fact>
";
    String fact = pattern.replace("$SRC", src.getName());
    fact = fact.replace("$DEST", dest.getName());
    fact = fact.replace("$PRIORITY", priority);
    bbmeansend = bbmeansend.concat(fact);
}

public void addBBContrib(ElementContainer src, ElementContainer dest, String value) {
    String pattern = "<t\t<fact>new TContrib("$SRC", "$DEST", "$VAL")</fact>
";
    String fact = pattern.replace("$SRC", src.getName());
    fact = fact.replace("$DEST", dest.getName());
    fact = fact.replace("$VAL", value);
    bbcontrib = bbcontrib.concat(fact);
}

/**
 * @param goal The 'why' for the dependency, our starting point.
 * @param dependum The dependum goal.
 * @param dependee The dependee Actor.
 */

private void addBBDependency(GoalContainer goal, String dependum, String dependee) {
    String pattern = "<t\t<fact>new TDependency("$SRC", "$DEPENDUM", "$DEPENDENT")</fact>
";
    String fact = pattern.replace("$SRC", goal.getName());
    fact = fact.replace("$DEPENDUM", dependum);
    fact = fact.replace("$DEPENDENT", dependee);
}
fact = fact.replace("$DEST", dependee);
bbdepend = bbdepend.concat(fact);
}/**
 * writes the plan contributions and the bodies of the real plans.
 */
private void writePlans() {
 String file_RealPlan = readFileAsString(sourcepath + "RealPlan_TMP.java");
 String realplanpattern = readFileAsString(sourcepath + "pattern_realplan.xml");
 for (PlanContainer plan : ad.planbase.values()) {
 // retrieve softgoal contributions
 for (SoftgoalContainer sg : plan.getContributions().keySet()) {
 addBBContrib(plan, sg, plan.getContributions().get(sg));
 }
 writeMeansEndLink(plan, realplanpattern);
 // /copy and rename the real plan body
 String newfile = file_RealPlan.replace("$PACKAGE", packagename);
 newfile = newfile.replace("TMP", plan.getName());
 writeFile(newfile, path + "RealPlan_" + plan.getName() + ".java");
 // TODO: add access to resources in plan body!
 }
}
/**
 * writes the dispatch plans (with bodies) for every child goal
 */
private void writeMeansEndLink(PlanContainer plan, String pattern) {
 String params="", results="", triggers="";
 for (GoalContainer meansEnd:plan.getMEGoals()) {
 params=params.concat("<goalmapping ref=""+meansEnd.getName()+"".param"/>");
 results=results.concat("<goalmapping ref=""+meansEnd.getName()+"".result"/>");
 triggers=triggers.concat("<goal ref=""+meansEnd.getName()+""/>");
 addBBMeansEnd(meansEnd, plan);
 }
 String adfplan = pattern.replace("$PARAMS", params);
 adfplan = adfplan.replace("$RESULTS", results);
 adfplan = adfplan.replace("$TRIGGERS", triggers);
 adfrealplans = adfrealplans.concat(adfplan);
}
private String readFileAsString(String filePath) {
 StringBuffer fileData = new StringBuffer(1000);
 try {
 BufferedReader reader = new BufferedReader(new FileReader(filePath));
 char[] buf = new char[1024];
 int numRead = 0;
 while ((numRead = reader.read(buf)) != -1) {
 fileData.append(new String(buf, 0, numRead));
 buf = new char[1024];
 } reader.close();
 } catch (FileNotFoundException e) {
 System.err.println("Error: file "+filePath+ " not found.");
 e.printStackTrace();
 } catch (IOException e) {
 e.printStackTrace();
 } return fileData.toString();
}/**
 * @param dispatchGoalPlan
 * @param string
 */
private void writeFile(String content, String filename) {
 PrintWriter file = null;
 try {
 file = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
 } catch (IOException e) {
 System.out.println("Error: Can't create file "+filename+ "!");
 System.exit(1);
 } file.println(content);
 file.close();
 public static void copyFile(String source, String dest) throws IOException {
}
public static void copyFile(File source, File dest) throws IOException {
    FileChannel in = new FileInputStream(source).getChannel();
    FileChannel out = new FileOutputStream(dest).getChannel();
    in.transferTo(0, in.size(), out);
    in.close();
    out.close();
}
Appendix C: The Code Generation Tool t2x

```java
private ArrayList<GoalContainer> goals; // means end plans
private ArrayList<PlanContainer> plans;
private ArrayList<String[]> dependencies;
private Const decomposition = Const.NONE;
private ArrayList<GoalContainer> parentlist;
/**
 * Creates a standard achieve goal with request plan.
 */
public GoalContainer(Goal goal, Const achieve) {
    super(goal);
    this.achieve = achieve;
    this.request = Const.NONE;
    goals = new ArrayList<GoalContainer>();
    plans = new ArrayList<PlanContainer>();
    parentlist = new ArrayList<GoalContainer>();
    dependencies = new ArrayList<String[]>();
}
public void setRequest(Const request) {
    this.request = request;
}
public PlanContainer addMEREalPlan(PlanContainer child) {
    plans.add(child);
    if (decomposition == Const.OR || decomposition == Const.ME) {
        assert decomposition == Const.ME; // otherwise there is an error elsewhere!
        // needed to add more triggering goals to one real plan
        child.addMEGoal(this);
    }
    return child;
}
public GoalContainer addDecomp(GoalContainer child) {
    goals.add(child);
    if (decomposition == Const.OR || decomposition == Const.ME) {
        assert decomposition == Const.OR; // otherwise there is an error elsewhere!
        // for this goals dispatch-plans are created (not needed for AND-goals)
        // (needed to add more triggering goals to one dispatch plan)
        child.addParent(this);
    }
    return child;
}
public void createDecomposition(Const decomp) {
    decomposition = decomp;
    if (decomp == Const.AND) {
    }
    else if (decomp == Const.OR || decomp == Const.ME) {
    }
}
public Const getDecomposition() {
    return decomposition;
}
/**
 * @return Returns the goals.
 */
public ArrayList<GoalContainer> getDecompGoals() {
    return goals;
}
/**
 * @return Returns the plans.
 */
public ArrayList<PlanContainer> getDecompPlans() {
    return plans;
}
/**
 * @return Returns all parent goals (This goal is part of the decomposition of a
 * parent goal).
 */
public ArrayList<GoalContainer> getParentGoals() {
    return parentlist;
}
private void addParent(GoalContainer gc) {
    parentlist.add(gc);
}
public void addDependency(String dependum, String dependee) {
    // dependencies.add(dep);
    dependencies.add(new String[] { dependum, dependee });
}
public ArrayList<String[]> getDependencies() {
```

134
/**
 * @param dep
 * @return The name of the dependee Actor.
 */
public String getActorFromDependency(String[] dep) {
    return dep[1];
}

/**
 * Returns the name of the dependum goal from a goal dependency.
 * @param dep
 * @return The name of the dependum goal
 */
public String getDependumGoalFromDependency(String[] dep) {
    return dep[0];
}
ADF templates

All templates needed to build an ADF file. Header, body and footer templates for an ADF file contain placeholders, filled by strings generated using the predefined patterns.

./t2x/producer/planskeletons/agentheader.xml
./t2x/producer/planskeletons/agentbody.xml
./t2x/producer/planskeletons/agentfooter.xml
./t2x/producer/planskeletons/pattern_dispatchANDplan.xml
./t2x/producer/planskeletons/pattern_dispatchplan.xml
./t2x/producer/planskeletons/pattern_event.xml
./t2x/producer/planskeletons/pattern_goal.xml
./t2x/producer/planskeletons/pattern_metagoal.xml
./t2x/producer/planskeletons/pattern_metaplan.xml
./t2x/producer/planskeletons/pattern_realplan.xml
./t2x/producer/planskeletons/pattern_requestplan.xml

<!----------------------------------------------------------------------------->
./t2x/producer/planskeletons/agentheader.xml
<!----------------------------------------------------------------------------->

<agent
xmlns="http://jadex.sourceforge.net/jadex"
xsi:schemaLocation="http://jadex.sourceforge.net/jadex
http://jadex.sourceforge.net/jadex-0.94.xsd"
name="$NAME"
package="$PACKAGE">
<imports>
<import>jadex.util.*</import>
<import>jadex.adapter.fipa.*</import>
<import>jadex.runtime.*</import>
<import>java.util.logging.*</import>
<import>t2x.util.plans.*</import>
<import>t2x.util.components.*</import>
</imports>

<beliefs>
<!------------------------------------------
./t2x/producer/planskeletons/agentbody.xml
------------------------------------------>
<beliefset name="goals" class="TGoal">
$BBGOALS
</beliefset>
<beliefset name="softgoals" class="TSoftgoal">
$BBSOFTGOALS
</beliefset>
<beliefset name="decomp" class="TLink">
$DECOMP
</beliefset>
<beliefset name="meansend" class="TLink">
$MEANSEND
</beliefset>
<beliefset name="contributions" class="TContrib">
$CONTRIB
</beliefset>
<beliefset name="dependencies" class="TDependency">
$DEPENDENCIES
</beliefset>
<!------------------------------------------
./t2x/producer/planskeletons/agentfooter.xml
------------------------------------------>
</beliefs>
</agent>
<!DOCTYPE events PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<events>

<goals>
   <!-- default values: metalevel reasoning enabled, exclude when tried. -->
</goals>

$GOALS
   <!-- The meta-level goals for choosing between plans (and goals). -->
$METAGOALS
</goals>

<plans>
   <!-- Initial plans for handling message requests. They have the task to create appropriate subgoals. They use waitqueues to store message events that arrived during the plan was busy. -->
</plans>

$REQUESTPLANS
   <!-- Plans triggered by a parent goal, used to dispatch a child goal. -->
<plan name="informChangePlan">
   <body>
      new InformChangePlan()</body>
   <trigger>
      <messageevent ref="inform_ChangeEnv"/>
   </trigger>
</plan>

$DISPATCHPLANS
   <!-- Plans associated to an AND-decomposed goal, used to dispatch all subgoals in sequence. -->
$DISPATCHANDPLANS
   <!-- Meta-Plans associated to their Metagoals, used to chose between alternative plans (and so between goals). -->
$METAPLANS
   <!-- Real Plans that hold the activity part of a capability and "do" the requested things. -->
$REALPLANS
</plans>

rels/producers/planskeletons/agentfooter.xml

<events>
   <!-- Specifies a request to achieve a goal (one per goal). -->
$EVENTS
   <!-- messages for informs on environment changes -->
<messageevent name="inform_ChangeEnv" direction="receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.INFORM</value>
   </parameter>
</messageevent>

<!-- FIPA-messages needed to communicate with other agents -->
<messageevent name="request" direction="send" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.REQUEST</value>
   </parameter>
</messageevent>

<messageevent name="agree" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.AGREE</value>
   </parameter>
</messageevent>

<messageevent name="failure" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.FAILURE</value>
   </parameter>
</messageevent>

<messageevent name="cancel" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.CANCEL</value>
   </parameter>
</messageevent>

<messageevent name="cancel" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.CANCEL</value>
   </parameter>
</messageevent>

<messageevent name="n_u" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.NOT_UNDERSTOOD</value>
   </parameter>
</messageevent>

<messageevent name="refuse" direction="send_receive" type="fipa">
   <parameter name="performativ" class="String" direction="fixed">
      $fipa.REJECT</value>
   </parameter>
</messageevent>
</events>
Appendix C: The Code Generation Tool t2x

<value>SFipa.REFUSE</value>
</messageevent>
</events>
<expressions>
<!-- All expressions are not changed during the automatic code generation -->
<!-- This query selects the first matching entry from the dictionary, whereby the parameter $word is compared to the first element of a belief set tuple. -->
<expression name="query_link">
  select $link.getGoal(1)
  from TLink $link in $beliefbase.decomp
  where $link.get(0).equals($component) order by $link.getPriority() desc
  <parameter name="$component" class="String"/>
</expression>
<expression name="query_ME_link">
  select $link.get(1)
  from TLink $link in $beliefbase.meansend
  where $link.get(0).equals($component)
  <parameter name="$component" class="String"/>
</expression>
<expression name="query_contributions">
  select $link
  from TContrib $link in $beliefbase.contributions
  where $link.get(0).equals($component)
  <parameter name="$component" class="String"/>
</expression>
<expression name="query_dependencies">
  select $link
  from TDependency $link in $beliefbase.dependencies
  where $link.getWhyGoal().equals($component)
  <parameter name="$component" class="String"/>
</expression>
</expressions>
<properties>
<!-- Only log outputs >= level are printed. -->
<property name="logging.level">Level.INFO</property>
<!-- The default parent handler prints out log messages on the console. -->
<property name="logging.useParentHandlers">true</property>
</properties>
</agent>
<!-----------------------------------------------------------------------------.t2x/producer/planskeletons/pattern_dispatchANDplan.xml------------------------------------------------------------------------------>
<plan name="dispatchANDGoalPlan_$NAME">
  <parameter name="param" class="String">
    <goalmapping ref="$NAME.param"/>
  </parameter>
  <parameter name="result" class="String">
    <goalmapping ref="$NAME.result"/>
  </parameter>
  <body>new ANDGoalPlan_$NAME()</body>
  <trigger>
    <goal ref="$NAME"/>
  </trigger>
</plan>
<!-----------------------------------------------------------------------------.t2x/producer/planskeletons/pattern_dispatchplan.xml------------------------------------------------------------------------------>
<plan name="dispatchGoalPlan_$CHILD">
  <parameter name="param" class="String">
    $PARAMS
  </parameter>
  <parameter name="result" class="String">
    $RESULTS
  </parameter>
  <body>new DispatchGoalPlan("$CHILD")</body>
  <trigger>
    $TRIGGERS
  </trigger>
</plan>
<!-----------------------------------------------------------------------------.t2x/producer/planskeletons/pattern_event.xml------------------------------------------------------------------------------>
<plan name="dispatchANDGoalPlan_$NAME">
  <parameter name="param" class="String">
    $PARAMS
  </parameter>
  <parameter name="result" class="String">
    $RESULTS
  </parameter>
  <body>new ANDGoalPlan("$NAME")</body>
  <trigger>
    $TRIGGERS
  </trigger>
</plan>
<messageevent name="request_$NAME" direction="receive" type="fipa">
  <value>$fipa.REQUEST</value>
</parameter>
<parameter name="performative" class="String" direction="fixed">
  <value>$NAME</value>
</parameter>
</messageevent>

<achievegoal name="$NAME">
  <parameter name="param" class="String"/>
  <parameter name="result" class="String" direction="out"/>
</achievegoal>

<metagoal name="meta_$NAME">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <trigger>
    <goal ref="$NAME"/>
  </trigger>
</metagoal>

<plan name="metaPlan_$NAME">
  <parameterset name="applicables" class="ICandidateInfo"/>
  <goalmapping ref="meta_$NAME.applicables"/>
  <parameterset name="result" class="ICandidateInfo" direction="out"/>
  <goalmapping ref="meta_$NAME.result"/>
  <body>
    new MetaPlan_$NAME()
  </body>
  <trigger>
    <goal ref="meta_$NAME"/>
  </trigger>
</plan>

<plan name="realPlan_$PLANNAME">
  <parameter name="param" class="String">
    $PARAMS
  </parameter>
  <parameter name="result" class="String">
    $RESULTS
  </parameter>
  <body>
    new RealPlan_$PLANNAME()
  </body>
  <trigger>
    $TRIGGERS
  </trigger>
</plan>

<plan name="requestPlan_$NAME">
  <body>
    new GoalRequestPlan("$NAME")
  </body>
  <trigger>
    <messageevent ref="request_$NAME"/>
  </trigger>
</plan>
Plan skeletons

All skeletons used by the code generator to write the Java files associated to the plans defined in an ADF.

./t2x/producer/planskeletons/ANDGoalPlan_TMP.java
./t2x/producer/planskeletons/DispatchGoalPlan.java
./t2x/producer/planskeletons/GoalRequestPlan.java
./t2x/producer/planskeletons/InformChangePlan.java
./t2x/producer/planskeletons/MetaPlan_TMP.java
./t2x/producer/planskeletons/RealPlan_TMP.java

package $PACKAGE;
import t2x.util.plans.BaseANDDispatchGoalPlan;
/**
* 
*/
public class ANDGoalPlan_TMP extends BaseANDDispatchGoalPlan {
    // for a custom implementation:
    // protected void goalsANDdispatch(String param) {...}
}

package $PACKAGE;
import t2x.util.plans.BaseDispatchGoalPlan;
public class DispatchGoalPlan extends BaseDispatchGoalPlan {
    public DispatchGoalPlan(String requestedGoal) {
        super(requestedGoal);
    }
}

package $PACKAGE;
import t2x.util.plans.BaseGoalRequestPlan;
import jadex.runtime.IMessageEvent;
public class GoalRequestPlan extends BaseGoalRequestPlan {
    public GoalRequestPlan(String requestedGoal) {
        super(requestedGoal);
    }
    @Override
    protected void goalsSuccess(String name, String param, Object result) {
        String cont = "Goal success with " + name + " \"" + param + "\": " + result;
        getLogger().info(cont);
        sendMessage(((IMessageEvent) getInitialEvent()).createReply("inform", cont));
    }
    @Override
    protected void goalFailure(String name, String param) {
        String cont = "Failure with goal " + name + "\": " + param;
        getLogger().info(cont);
        sendMessage(((IMessageEvent) getInitialEvent()).createReply("failure", cont));
    }
    @Override
    protected void requestFailure(String request) {
        String cont = "Request format not correct, needs: #" + "action param";
        getLogger().info(cont);
        sendMessage(((IMessageEvent) getInitialEvent()).createReply("failure", cont));
    }
    @Override
    protected void goalAgree(String name, String param) {
        sendMessage(((IMessageEvent) getInitialEvent()).createReply("agree", param));
    }
}
package $PACKAGE;
import t2x.util.plans.BaseInformChangePlan;

public class InformChangePlan extends BaseInformChangePlan {
    public InformChangePlan() {
        super();
    }
}

package $PACKAGE;
import t2x.util.plans.AMetaPlan;

public class MetaPlan_TMP extends AMetaPlan {
    // protected ICandidateInfo selection(ICandidateInfo[] applicables) {
    //     return applicables[0];
    // }
}

package $PACKAGE;
import t2x.util.plans.BaseRealPlan;

public class RealPlan_TMP extends BaseRealPlan {
    @Override
    public void body() {
        String param = (String) getParameter("param").getValue();
        String resource = (String) useResource("RESOURCE");
        String capabilityName = getPlanName();
        requestCapability(capabilityName, param);
        getParameter("result").setValue(param);
        produceResource("RESOURCE", result);
    }
}

"}
References


143
References


