Using three AOSE toolkits to develop a sample design

Scott A. DeLoach

Kansas State University, USA
E-mail: sdeloach@ksu.edu

Lin Padgham*

RMIT University
Melbourne, Australia
Fax: +61 3 9662 1617
E-mail: lin.padgham@rmit.edu.au
*Corresponding author

Anna Perini and Angelo Susi

Fondazione Bruno Kessler – IRST
Trento, Italy
E-mail: perini@itc.it
E-mail: susi@fbk.eu

John Thangarajah

RMIT University
Melbourne, Australia
Fax: +61 3 9662 1617
E-mail: john.thangarajah@rmit.edu.au

Abstract: At the *8th Agent-Oriented Software Engineering Workshop*, the developers of tools supporting three popular agent-oriented methodologies (Tropos, Prometheus and Organization-based Multiagent Systems Engineering (O-MaSE)) demonstrated their tools using a common multi-agent system design case study: the Conference Management System. The methodologies are representative of the state-of-the-art in agent-oriented software methodologies, as they are some of the earliest and most mature agent-oriented methodologies currently available. The paper briefly summarises the three methodologies and their associated tools and then works through the analysis, architectural design and detailed design phases of the Conference Management system case study using each methodology and tool. The paper compares the models and concepts used during each phase and provides a discussion on the similarities and differences between them.

Keywords: agent design toolkits; Prometheus; Tropos; Organization-based Multiagent Systems Engineering; O-MaSE; agent-oriented software engineering; AOSE.

Reference to this paper should be made as follows: DeLoach, S.A., Padgham, L., Perini, A., Susi, A. and Thangarajah, J. (2009) 'Using three AOSE toolkits to develop a sample design', *Int. J. Agent-Oriented Software Engineering*, Vol. 3, No. 4, pp.416–476.

Biographical notes: Scott A. DeLoach is an Associate Professor in the Computing and Information Sciences Department at Kansas State University, USA. His research focuses on methods and techniques for the analysis, design and implementation of complex adaptive systems, which have been applied to both multi-agent and cooperative robotic systems. Dr. DeLoach is best known for his work in agent-oriented software engineering. He is the creator of the Multiagent Systems Engineering methodology (MaSE), its follow-on Organization-based Multiagent Systems Engineering methodology (O-MaSE), and the associated agentTool analysis and design tool. He has more than 50 refereed publications and has advised over 25 graduate students. Dr. DeLoach came to Kansas State University after a 20-year career in the US Air Force.

Lin Padgham is a Professor in Artificial Intelligence at RMIT University, Australia. She has spent more than ten years researching intelligent multi-agent systems and has developed (with colleagues) the Prometheus design methodology for building agent systems, and co-authored the first detailed book (published 2004) on a methodology for building multi-agent systems. In 2005, the supporting tool for this methodology, the Prometheus Design Tool, won the award for best demonstration at AAMAS'05. Padgham serves on the editorial board of *Autonomous Agents and Multi-Agent Systems*, and was Program Co-Chair for AAMAS 2008.

Anna Perini is a Research Leader at the Software Engineering group of FBK-IRST CIT (http://se.fbk.eu). Her main research interests are in agent-oriented software engineering and requirements engineering. She participates in the Programme Committee of international conferences and workshops and acts as a Reviewer for major journals in these areas. She has led several research and industrial projects. She taught Software Engineering at the University of Trento (1999–2006) and gave tutorials on Agent-Oriented Software Engineering (Agentlink Summer School EASSS 2004, UPC Master Course 2007, IEEE Requirements Engineering Conf., 2008).

Angelo Susi is a Researcher in the Software Engineering group at FBK-IRST in Trento, Italy. His research interests are in the areas of requirements engineering, agent-oriented software engineering and machine learning applied to software requirements management. In particular, Susi is active in the development of the Tropos Agent-Oriented Software Engineering methodology and of the requirements prioritisations methodology CBRank, based on machine learning techniques. He participates in the programme committees of international conferences and workshops, such as AAMAS, ACM SAC and ICSOC, and he serves as a reviewer for journals in these areas.

John Thangarajah is a Research Fellow at the School of Computer Science at RMIT University, Australia. He completed his PhD in 2004, which addressed the issue of managing the interactions between the goals of an agent. His research expertise and interests at present are in the areas of agent reasoning and agent-oriented software engineering. He has over 20 refereed international publications in these two main research areas. He has been involved in developing the Prometheus Agent development methodology and the

Prometheus Design Tool, which has won an award at an international conference. He has served on the programme committee of top international conferences and is also the Co-Chair of PROMAS 2009.

1 Introduction

Many Agent-Oriented Software Engineering (AOSE) methodologies have been proposed over the last five to ten years (Bergenti *et al.*, 2004; Henderson-Sellers and Giorgini, 2005). This has motivated research on how to compare and evaluate these methodologies, with the purpose of pointing out differences and commonalities and of giving criteria for selecting the most appropriate methodology, for a given development scenario (Dam and Winikoff, 2003; Henderson-Sellers and Giorgini, 2005). Presentation of common examples in the various approaches can also provide assistance in understanding both the commonalities and the differences between approaches.

At the *8th Agent-Oriented Software Engineering Workshop*, held in Hawaii in May 2007, developers of tools supporting three fairly well developed methodologies, *Tropos* (Bresciani *et al.*, 2004), Prometheus (Padgham and Winikoff, 2004) and Organization-based Multiagent Systems Engineering (O-MaSE) (DeLoach, 2001), presented their tools (*TAOM4E*, Prometheus Design Tool (PDT) and *aT*^{3} respectively) by demonstrating their use on a popular multi-agent system design case study: the Conference Management System.[1] The Conference Management System case study was first proposed in (Ciancarini *et al.*, 1998) and has been used several times in the literature as a motivating example (Ciancarini *et al.*, 1999; DeLoach, 2002; Zambonelli *et al.*, 2001). The Conference Management System case study has gained popularity as it is suitable for illustrating a variety of multi-agent system analysis and design issues. Presenters were referred to the Conference Management System as described in (DeLoach, 2002), as a focus for their design and tool presentations. Quoting from (DeLoach, 2002):

> "The Conference Management System is an open multiagent system supporting the management of various sized international conferences that requires the coordination of several individuals and groups. There are four distinct phases in which the system must operate: submission, review, decision and final paper collection. During the submission phase, authors should be notified of paper receipt and given a paper submission number. After the submission deadline has passed, the Programme Committee (PC) has to review the papers by either contacting referees and asking them to review a number of the papers or reviewing them themselves. After the reviews are complete, a decision on accepting or rejecting each paper must be made. After the decisions are made, authors are notified of the decisions and are asked to produce a final version of their paper if it was accepted. Finally all final copies are collected and printed in the conference proceedings."

This paper presents the application of the three methodologies and their supporting tools on the Conference Management System over three different design phases: Analysis, Architectural Design, and Detailed Design. The goal of the paper is to clearly demonstrate the similarities and differences between the methodologies and their tools as

well as to showcase the general state-of-the-art in agent oriented analysis and design. Individual papers presenting each of the approaches are also published in the AOSE workshop post-proceedings (Luck and Padgham, 2008).

The three methodologies considered in this work are representative of state-of-the-art AOSE approaches. They are some of the earliest agent-oriented methodologies, each of which developed from a different perspective. O-MaSE (previously MaSE) descended from an object-oriented background and adapted the techniques and models to the agent paradigm. Prometheus arose from significant experience in developing BDI agent systems and assisting both students and companies in understanding the principles of designing such systems. This resulted in a set of models to capture the analysis and design of intelligent agent systems, along with processes for developing these models. *Tropos* adopted a requirements driven approach, building on goal oriented approaches for domain and requirements analysis and adapting their analysis methods to the design of agent-based systems. Each of these methodologies has evolved since their original definitions and although mutual influences can be observed, their roots are reflected in differences in the way some of the agent abstractions are used and in the scope of the supported process. Both similarities and differences are identified throughout the paper.

It is generally agreed that before any design methodologies will be widely used and accepted in industrial settings, tools that support those methodologies must be readily available and have reached a certain level of maturity. Therefore, a common objective underlying the work presented here is to provide tool-supported frameworks for the selected methodologies in order to encourage their adoption by industry. We believe that presenting them side-by-side using the same case study will be beneficial in pointing out common approaches and techniques, while also comparing their unique perspectives in addressing common problems.

In the following section we provide a brief overview of the three tools covered in the paper. We then work through the Analysis, Architectural Design and Detailed Design phases of the Conference Management case study for each of the three methodologies and tools. To complete our coverage of the tools, we end by describing additional features for each, followed by a brief conclusion.

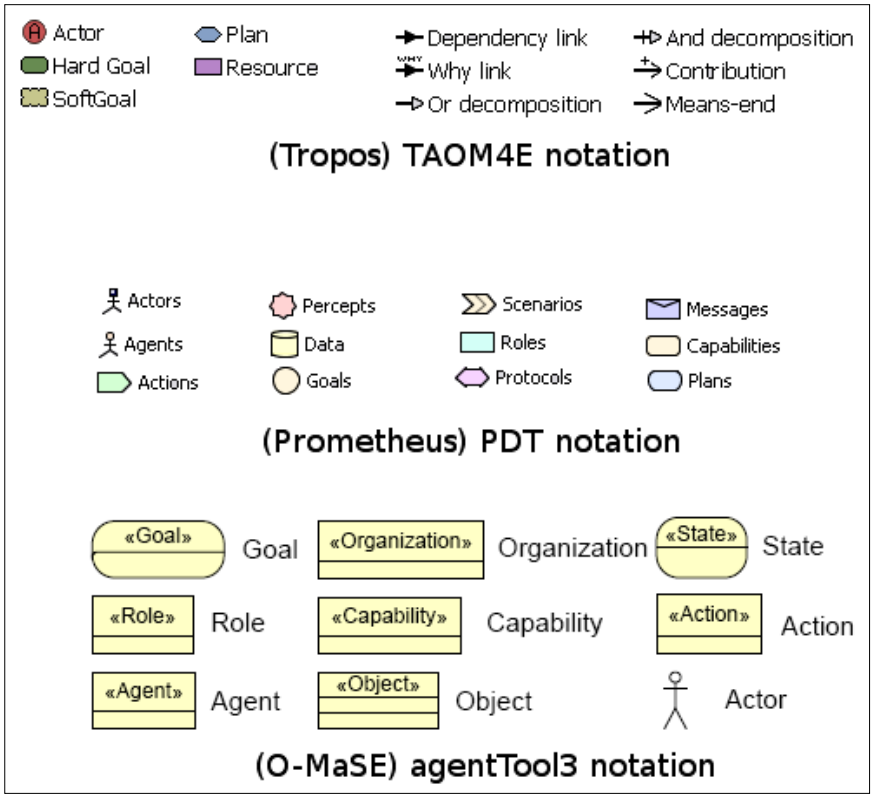
2 Tools overview

All the toolkits covered here are relatively well developed and are publicly available for download via the internet. Although the developers are working towards a consistent notation, at the current point in time each toolkit has a different notation. We identify the symbols used for the graphical models in each system in Figure 1, which will serve as a legend for many of the figures throughout the paper.

As can be seen, although the symbols are different, there is substantial similarity in concepts used, although, as we will see, there are some differences in the precise meaning of some of these concepts. All tools use the concepts of actors and goals, with specific symbols for these. PDT and aT^3 both have agents, roles, capabilities and actions with meanings similar to each other. *TAOM4E* uses the actor symbol and concept to cover agents, conceptualising these as system actors. aT^3 and PDT also both have protocols and messages, although these are shown as arrows in aT^3 and as graphical entities in PDT. The ‘Resource’ of *TAOM4E*, ‘Data’ of PDT and ‘Object’ and ‘State’ of aT^3 all allow for

representation of domain information and entities that are outside the agent paradigm. *TAOM4E* and aT^3 both have a variety of relationships, which they capture by arrows of different types. In PDT some of this information is not captured, although much is captured by other means as will be seen as we describe the design process. *TAOM4E* is unique in modelling soft goals, PDT is unique in modelling scenarios and aT^3 is unique in modelling organisations.

Figure 1 Graphical symbols used within each of the toolkits (see online version for colours)



Each of the tools are presented briefly below, while a comparison of the similarities and differences, and strengths and weaknesses of each can be found in Sections 3.4, 4.4, 5.4 and 6.4.

2.1 Tropos methodology and the TAOM4E Tool

The goal of the *Tropos* methodology is to support the agent paradigm and its associated mentalistic notions throughout the entire software development life cyle, from the early phases of requirements analysis through implementation. As its starting point, *Tropos* uses a conceptual modelling language based on the *i** framework (Yu, 1995). The basic concepts of the modelling language include actors, goals, plans and goal achievement dependencies. In addition, UML/AUML activity and sequence diagrams are used to support detailed design.

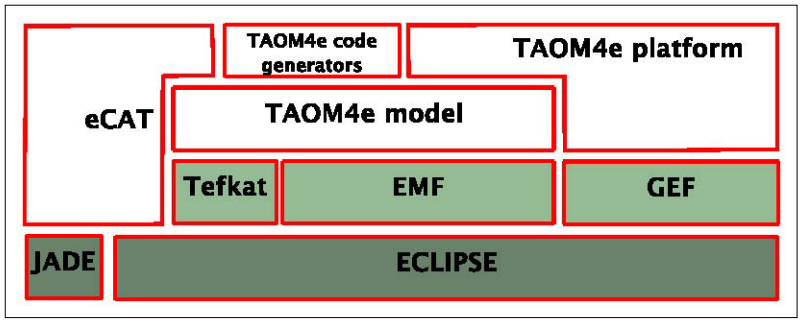
The *Tropos* development process includes five phases as shown in Table 1: Early Requirements, Late Requirements, Architectural Design, Detailed Design and Implementation. The Early Requirements phases focuses on the understanding of the problem domain prior to the introduction of the system, while the Late Requirements phases is for analysing the actual system-to-be. In the Architectural Design phase the system's global architecture is defined in terms of subsystems while in the Detailed Design phases the internals of the individual agents are specified. Finally, during the Implementation phase, code is generated according to the detailed design specifications. Table 1 shows the modelling activities performed and the artefacts produced during each phase of the CMS case study, which is a slight adaptation of the general process described in (Penserini *et al.*, 2007b).

Table 1 The development process in TAOM4E by phases, activities and work products

Phase	*Modelling activity and concepts*	*Work products*
Early Req. (ER)	ER actor modelling. Concepts: *actor*, *dependency*	ER actor diagram – Figure 8
	ER goal/plan modelling. Concepts: *goal*, *plan*, *decomposition*, *means_end*, *contribution*, *constraint*, *annotation*	ER goal diagram – Figure 9
Late Req. (LR)	LR actor modelling. Concepts: system-to-be, *actor*, *dependency*	LR actor diagram – Figure 10
	LR goal/plan modelling. Concepts: system-to-be, *goal*, *plan*, *decomposition*, *means_ends*, *contribution*, *constraint*, *annotation*	LR goal diagram –Figure 11
Archit. Design (AD)	AD actor modelling. Concepts: *agent* and agent *roles*	Agent/Role AD actor diagram – Figure 22
Detailed Design (DD)	DD capability/plan modelling. Concepts: sub-plans of each agent	DD agent goal diagram – Figure 30
	DD capability modelling: specification of the dynamic part via UML 2.0 sequence and activity diagrams	Capability's activity and sequence diagrams – Figure 31
	DD capability modelling: specification BDI structures	Computational representation of BDI concepts – Figure 36
Implement.	Code generation	MAS structure

Each *Tropos* phase is supported by the *TAOM4E* tool.[2] *TAOM4E* is implemented as an Eclipse[3] plug-in and extends the EMF, GEF and Tefkat plug-ins, as shown in Figure 2. The Eclipse Modelling Framework (EMF) plug-in[4] is a modelling framework for building applications based on an underlying model specified in XMI, which in this case is the *Tropos* metamodel. The Graphical Editing Framework (GEF) plug-in[5] provides facilities to build a graphical editor based on the *Tropos* metamodel. Finally, the Tefkat plug-in[6] is used to transform top-level plans and their decompositions into UML activity diagrams. The activity diagrams can be edited using any UML2 editor and further refined by sequence diagrams to define the requisite agent communication protocols.

Figure 2 *TAOM4E* architecture (see online version for colours)



As stated above, the *Tropos* metamodel was implemented using the Eclipse EMF. Figure 3 is a UML depiction of the portion of the *Tropos* metamodel that captures the *Tropos* dependency relationship and its four arguments: depender, dependee, dependum and why. The *depender* and the *dependee* are actors while the *dependum* is a goal. An optional argument, called *why*, can be either a goal, plan or resource.

Figure 3 A fragment of the *Tropos* metamodel implemented in *TAOM4E* specified in a UML class diagram

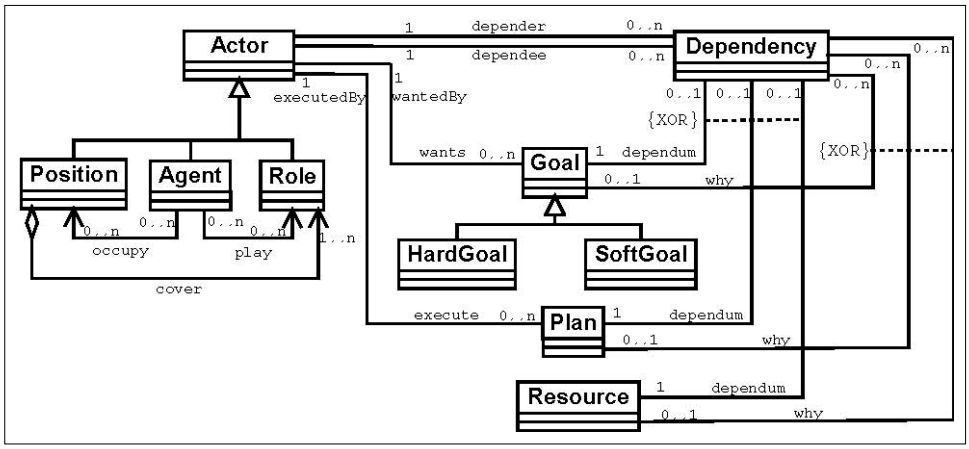
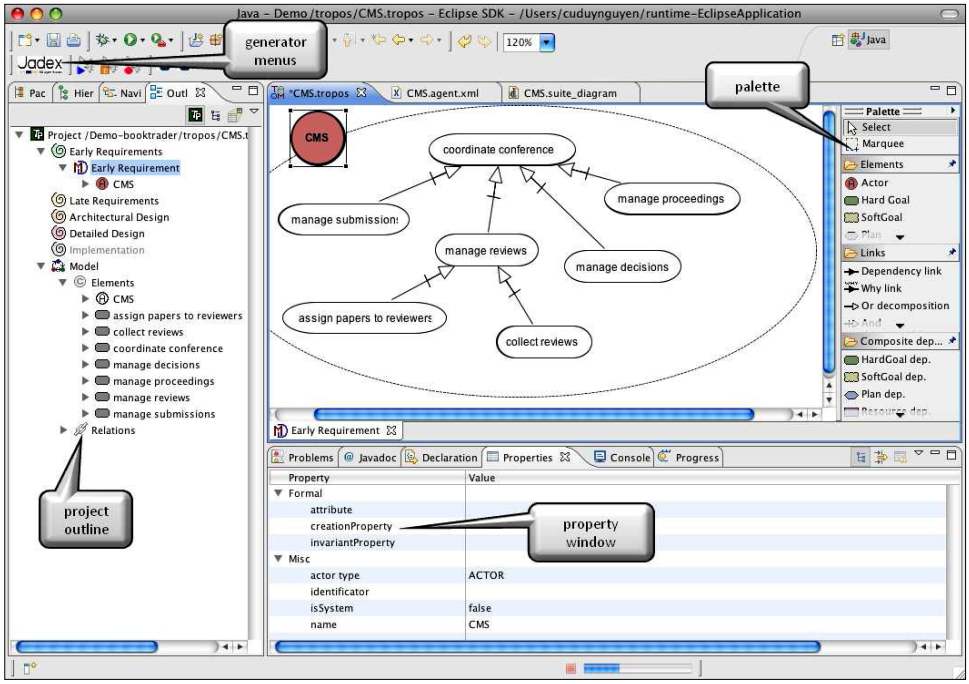


Figure 4 shows the *TAOM4E* user interface. The interface has five main components: a set of menus, the diagram editor, the editor palette, the properties window and the project browser. *Tropos* models are created using the various views available in the diagram editor. Actors, goals and dependency relationships are created by selecting their icons from the palette and placing them onto the current diagram. Dependency relationships define the social structure of the domain under analysis, which is represented by the *actor diagram*'s global view of the model. The individual actor perspective on how to achieve its goals by means of plans and resources is represented in the actor's *goal diagram*. By double-clicking on the actor, the editor dynamically switches between the global view and the individual actor views. Model element properties, such as goal instance, creation, fulfilment and invariant conditions, can be specified via the property panel window. Such invariants can be automatically mapped

into a formal *Tropos* specification and validated using model checking by the T-Tool (Fuxman *et al.*, 2001). The project browser provides navigation functions to explore a model by model element type or process phase.

Figure 4 A screenshot of the *TAOM4E* tool (see online version for colours)



TAOM4E also includes a suite of code generators – *UML2JADE*, *t2x* and *Tropos2UML* – that produce code skeletons for either *JADE* or *Jadex* agents. The code is generated directly from the UML detailed design specifications, detailed design artefacts and the *Tropos* goal model. *Tropos2UML* generates UML activities diagrams from *Tropos* goal models, while *UML2JADE* generates *JADE* agent code skeletons from UML activity and sequence diagrams, which capture *Tropos* plans (capabilities). Details on *TAOM4E* code generation is given in (Penserini *et al.*, 2006).

2.2 *Prometheus methodology and the Prometheus design tool*

The Prometheus methodology has been developed over more than ten years as a result of working with industry partners who are building agent systems and agent development tools. It has also been continually refined and developed through teaching both undergraduate and postgraduate students as well as running industry seminars. The tool support has arisen out of the need to provide this at a reasonable level for building even relatively small systems. For larger systems it is essential in order to maintain consistency even of such simple things as naming.

The PDT is freely available for download[7] and runs (currently) under Java 1.5. PDT is a stand-alone application but is also available as an Eclipse plug-in. Once this plug-in is installed, PDT can be selected as the design option within Eclipse.[8]

The design phases of the Prometheus methodology are summarised in Table 2 which shows the three main phases of design and for each phase the associated design tasks and the design diagrams available in the tool to support the task. Figure 5 provides a more detailed overview of the three main phases of design and associated design artefacts. Some design artefacts are regarded as final and part of the design documentation. Others are simply part of a process to help designers develop and refine their design.

Table 2 The development process in PDT by phases, activities and work products

Phase	*Modelling activity and concepts*	*Work products*
System specification	Identify actors, percepts, actions and scenarios	Analysis overview
	Develop scenarios	Scenario specifications
	Model goals	Goal overview diagram
	Model roles and their associated goals	Roles diagram
Architectural design	Model data and the roles that access the data, Identify the agents and assign roles to the agents, define any associations between the agents	Agent descriptors, agent-role grouping diagram
	Define the protocols for communication between the agents, ensure that the actions and percepts are handled appropriately by the agents	Protocol diagrams and descriptors, message descriptors
	Define any shared data entities	System overview diagram
Detailed design	Define the internals of the agent in terms of plans and capabilities, and the percepts or messages that are handled by them and the actions or messages produced	Agent descriptors, Plan descriptors, Event/Message descriptors, Capability descriptors, Agent overview diagram
	Define the internals of the capability in terms of plans and sub-capabilities similar to the agent overview diagram	Capability overview diagram, plan descriptors, event/message descriptors, capability descriptors

The tool supports four main kinds of design activities:

1 Development of graphical models of the system structure. These are listed in the upper left pane of the tool (see Figure 6) and developed in the upper right pane. These models are shown as the various overviews in Figure 5.

2 Development of process descriptions for scenarios and protocols. These are developed in pop-up windows activated from the Entities menu in the toolbar, where the specific scenario/protocol can be selected. Alternatively the pop-up window can be accessed by clicking on the relevant entity within the diagram pane. These entities capturing system dynamics are shown on the left hand side in Figure 5.

3 Development of detailed descriptors for each entity, which are in the bottom right pane of the tool (see Figure 6) and consist of a mix of free text fields and structured fields. These are (mostly) shown on the right hand side in Figure 5.

4 Consistency checking between models, which is activated by selecting this from the Tools menu on the toolbar. The tool maintains consistency automatically to a considerable extent, by such things as automated insertion of links in the system overview diagram, based on the message passing as specified in the protocol. However, more global consistency checking is done on request and includes such things as ensuring that all data are both used and produced somewhere in the design, and that all messages are used in some communication.

Once a design is produced, the code generation option on the 'Tools' menu produces skeleton JACK code based on the PDT models. The developer can iterate between design model changes (which are then propagated into the code) and development of code fragments within the skeleton framework. Others have used the framework to produce Jadex code (mentioned in Sudeikat *et al.*, 2004) as well as proof of concept 3APL code (Jayatilleke, 2007). The Tools menu also supports production of a design document containing diagrams of key system models, as well as a data dictionary and descriptions of all entities. This design document can be customised by selecting which figures and entities to include. It is produced as an HTML document and is hyperlinked. The developer can then further develop this document as desired.

Figure 5 Overview of the phases and artefacts of the Prometheus methodology (see online version for colours)

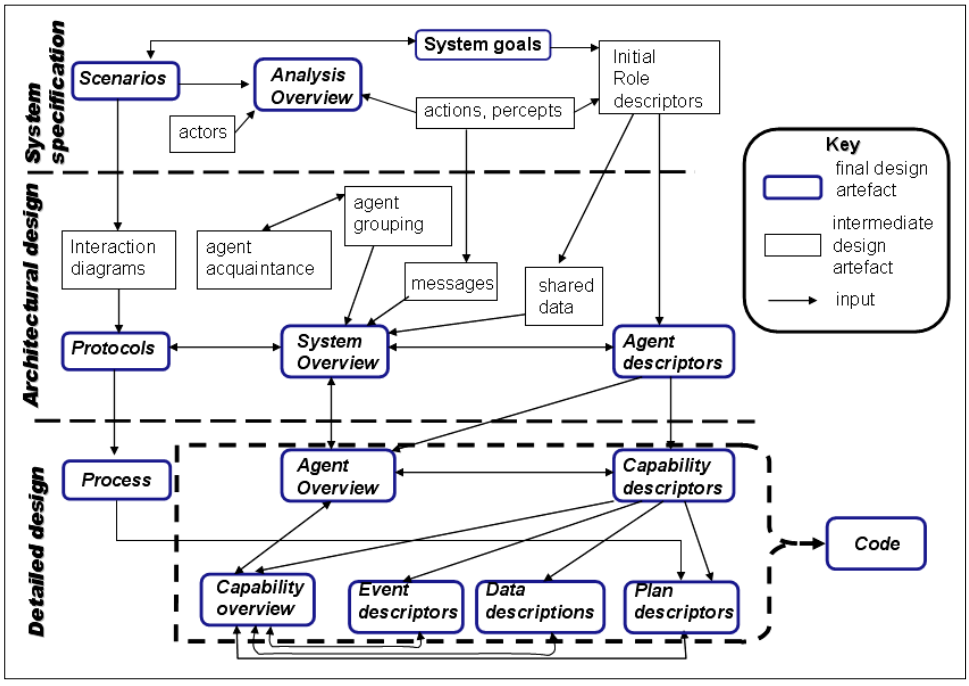
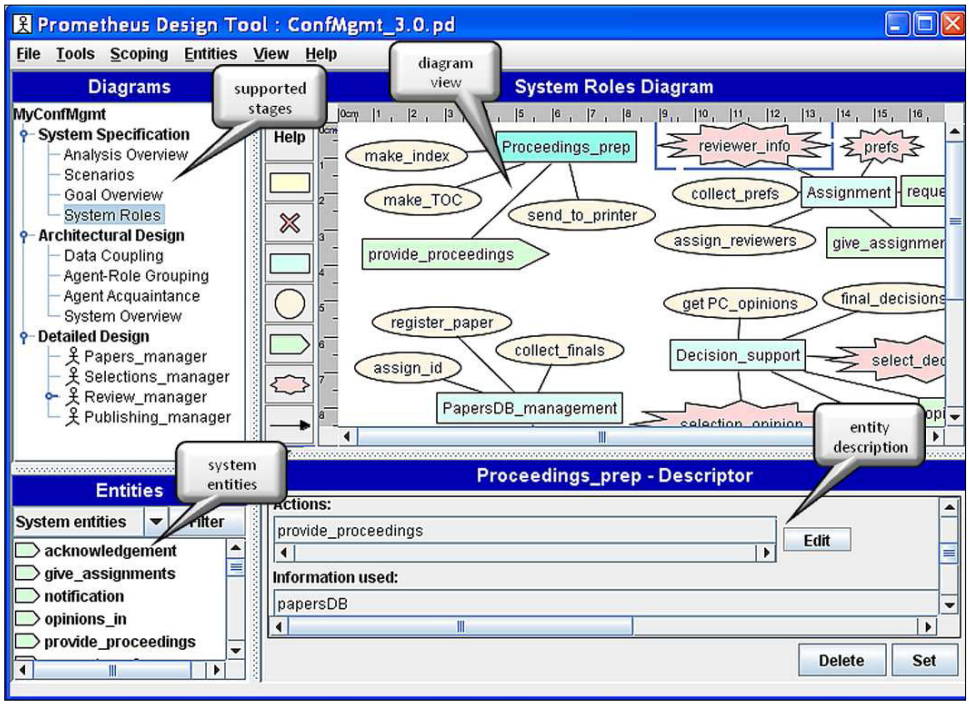


Figure 6 Overview of PDT (see online version for colours)



One of the features of PDT is that in addition to the user selected consistency checking, available from the Tools menu, it does enforce a degree of consistency between models. Entities introduced in one model or stage, are often automatically propagated to other models or stages where appropriate. This assists the developer in maintaining a consistent and coherent design.

There are a number of additional tools that interface to the models produced by PDT. These include a tool for automated unit testing of plans, events and beliefs (Zhang *et al.*, 2007), as well as a tool for runtime debugging based on PDT models (Poutakidis *et al.*, 2002; Padgham *et al.*, 2005). The testing tool is in the process of also being integrated into PDT. The CAFnE toolkit (Jayatilleke *et al.*, 2005b) is an extension to PDT that requires more detailed model based specification but then automatically produces complete executable code (as opposed to skeleton code which must be augmented to produce a functioning system). A methodology for designing social institutions using the Islander tool has also been developed to integrate with Prometheus (Sierra *et al.*, 2007) and the interfacing of the two toolkits is in process.

2.3 O-MaSE methodology and agentTool III

O-MaSE actually defines a framework, whose goal is to allow the designers to construct custom agent-oriented methodologies based on a set of method fragments, all of which are based on a common metamodel. To achieve this, O-MaSE is defined in terms of a metamodel, a set of method fragments and a set of guidelines. The O-MaSE metamodel

defines a set of analysis, design and implementation concepts and a set of constraints between them (Garcia-Ojeda *et al.*, 2007). The method fragments define how a set of analysis and design products may be created and used within the framework. Finally, guidelines define how the method fragment may be combined to create a complete instance of the O-MaSE methodology. A full treatment of these topics is beyond the scope of this paper; thus an overview of the phases, activities, tasks and work products currently supported by O-MaSE is shown in Table 3. Process designers select tasks and work products most appropriate to their situation and then verify that they form a O-MaSE compliant process by checking the chosen task against the O-MaSE process constructions guidelines.

Table 3 The development process in agentTool III (aT^3) by phases, activities and work products

Phase	*Modelling activity and concepts*	*Work products*
Requirements	Model goals	AND/OR goal tree
	Goal refinement	GmoDS model
	Model organisation interfaces	Organisation model
Analysis	Model roles	Role model
	Define roles	Role description document
	Model domain	Domain model
	Model agent classes	Agent class model
	Model protocol	Protocol model
	Model plans	Agent plan model
Design	Model policies	Policy model
	Model capabilities	Capabilities-action model
	Model actions	Capabilities-action model
Implementation	Code generation	Source code

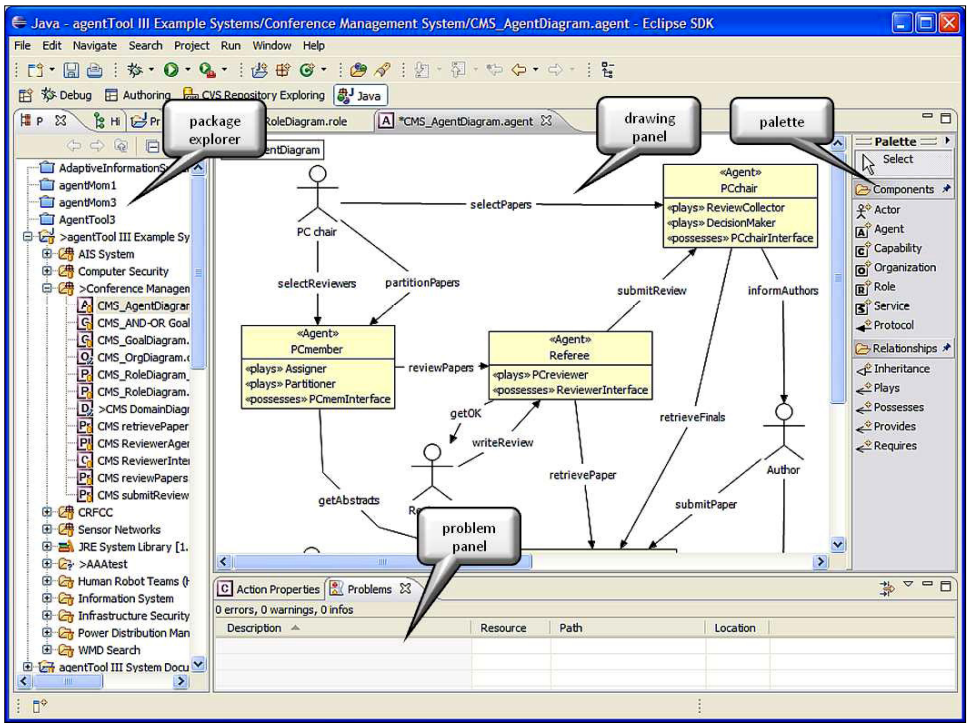
The O-MaSE methodology is supported by agentTool III (aT^3) development environment. aT^3 is based on agentTool 1 and 2, which supported the original MaSE methodology. agentTool was originally written as a standalone Java toolkit that allowed users to analyse and design multi-agent systems. The original agentTool supported protocol verification, semi-automatic analysis to design transformations and code generation. The aT^3 project webpage[9] contains the latest version of aT^3 for download as well as tutorials, documentation, and examples.

aT^3 is a completely new development from the original agentTool and, like *TAOM4E* and PDT, aT^3 is being developed as a set of Eclipse plugins. Firstly, we have developed a plugin for each O-MaSE model. Currently there are eight O-MaSE models implemented including the Goal Model, Organisation Model, Role Model, Agent Class Model, Protocol Model, Plan Model, Capability-Action Model and Domain Model. We are also currently developing the Policy Model, which will allow users to specify policies that apply to an organisation. There is also a single *Core* plugin that implements the O-MaSE metamodel entities and handles reading and writing of these entities to files and is used

by all aT^3 plugins. This architecture of multiple plugins supports the goal of allowing O-MaSE to be highly tailorable and extensible. None of the models are mandatory and new models may be easily incorporated into the tool.

A screen shot of the aT^3 is shown in Figure 7. On the left side of the screen, the Eclipse Package Explorer allows the user to organise and store O-MaSE models in projects. Generally, subdirectories within projects refer to sub-organisations in the system design, thus the Package Explorer file structure mimics the hierarchical structure of the system. The model shown is an Agent Class Diagram. The icons shown in the Palette on the right side of the screen show the valid components and relations that may be added to the model. To add a component to the model, users simply click on the component in the Palette and then click where they want to place the component in the model. Once the component has been placed in the model, it may be edited or moved to another location. The protocol components are slightly different in that they are added between two actors or agents. To add a protocol, the user first clicks on the protocol icon in the Palette and then on the two actors/agents that participate in the protocol. After placing the protocol, the name may be edited. To add relationships between model components, the user also clicks on the desired relationship in the Palette and then click on two components already in the model. Relationships have fixed names that may not be edited.

Figure 7 aT^3 agent class diagram (see online version for colours)



The notation used in the aT^3 models is very simple and consistent. Model components are generally represented as a box with several compartments. The top compartment specifies the type of component and the component name. Thus, in Figure 7, agent classes are represented as boxes with an ≪Agent≫ type. In the O-MaSE notation, guillemets are

used to enclose *type designators* and should not be confused with UML stereotypes. Directly below the type designation is the name of the agent class. Two unique component types are external actors and protocols. External actors are represented using a stick figure with the name of the actor directly below the figure, while a protocol is represented as an arrow between two components (*e.g.*, roles, agents, organisations). The name on the arrow is editable and represents the name of the protocol, which can be defined in detail via a protocol model. (A similar arrow notation is used in the protocol diagram to represent individual messages and in the goal diagram to represent individual events.) Relations between components are represented using traditional object-oriented notation for similar concepts such as inheritance and aggregation, and open headed arrows with fixed labels for O-MaSE specific relationships. In Figure 7, the ≪plays≫ arrow is used to define which agents can play which roles, the ≪possesses≫ arrow is used to define which agents possesses which capabilities, the ≪requires≫ arrow is used to define which roles require which capabilities, and the ≪provides≫ arrow is used to define which organisations, roles or agents provide which services.

As shown in Figure 7, aT^3 also supports embedding of relations to simplify the graphical layout of models. For instance, the `PCmember` agent class has two embedded relations: ≪plays≫ `Assigner` and ≪plays≫ `Partitioner`. This embedding represents the situation where the model also contains two additional roles and relationships between the `PCmember` and those two roles. In aT^3, the user may toggle between the embedded mode and the full mode, which shows all model components and relations explicitly.

3 Analysis

In developing any substantial software system, it is necessary to develop a relatively detailed understanding of the system to be produced and to ensure that there is general agreement between the builders of the system and those paying for its development. The analysis stage covers processes with a variety of names including *requirements analysis* and *system specification*. It is assumed here that a key aspect of this stage is to refine and develop a somewhat detailed, documented and agreed understanding of what the system to be built is intended to do.[10] Different approaches place different emphases on various aspects of this stage and this is also reflected in differences between the three toolkits covered here. We describe the support given by each of them for this stage and then discuss some of the similarities and differences.

3.1 Tropos and TAOM4E

This phase of development is broken down into two phases in *Tropos*: *Early Requirements* and the *Late Requirements*, each with its own respective model. The *Early Requirements* phase analyses the domain 'as is' while *Late Requirements* analyses the same domain once the system-to-be has been introduced.

The two requirements models are each an instantiation of the *Tropos* metamodel. In the Early Requirements model, the actors represented are the stakeholders; however, in the Late Requirements model, an instance of the system-to-be is inserted, which is represented as an actor. To focus on a specific part of the model under development,

Tropos models are visualised using various diagrams, each of which includes a subset of the current model elements and provides a different perspective. As introduced in Section 2.1, a *Tropos* model can be viewed from two different perspectives: the actor diagram, which defines the global network of dependencies among actors, and the goal diagram, which is the perspective of a single actor. To help guide the development of the Early Requirements model, questions such as *Who are the stakeholders in the domain?*, *What are stakeholder goals and how are they related to each other?* and *What are their strategic dependencies between actors for goal achievement?* are generally asked.

3.1.1 Early requirements

The goal of this phase is the production of the Early Requirements model and the associated set of actor and goal diagrams. Figures 8 and 9 show the actor and goal diagram views of the Conference Management System Early Requirements model. The main stakeholders in the CMS domain include paper authors, the conference programme committee, the programme committee chair, paper reviewers and the proceedings publisher. The stakeholders are represented as the actors Author, PC, PC Chair, Reviewer and Publisher. Next, stakeholder goals are identified. Based on domain information, the analyst must determine whether or not each goal is achievable by the actor itself or if the actor must delegate it to another actor. Goal delegation reveals a dependency relationship between actors, such as the dependency between Author and PC for the achievement of the Publish proceedings goal as shown in Figure 9. An analogous analysis is carried out for tasks and resources, according to the *Tropos* process described in (Giorgini *et al.*, 2008).

In practice, the Early Requirements model is built by creating an Actor Diagram in *TAOM4E* and adding actors, goals, *etc.*, into the model via the editor, as shown in Figure 8. Circles represent actors, ovals the goals and rectangles the resources, while the arrows linking pairs of actors, via a resource or goal, capture dependencies between the two actors for the goal achievement or resource usage.

Figure 8 Early requirements actor diagram (see online version for colours)

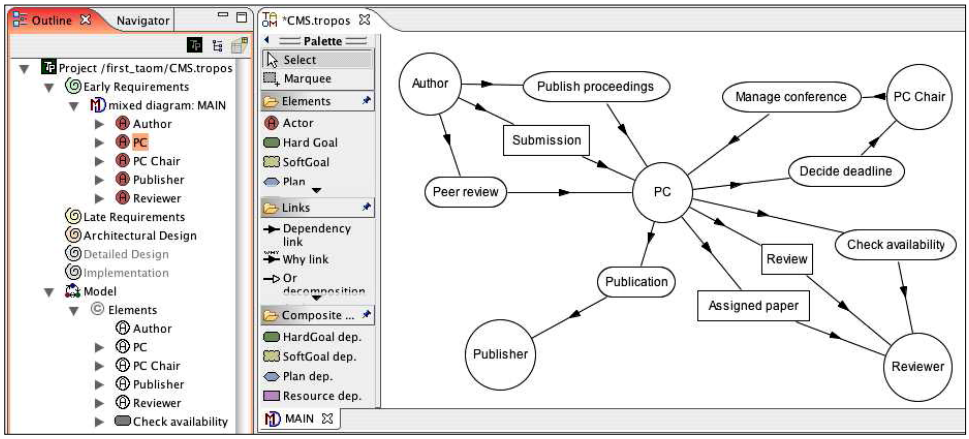
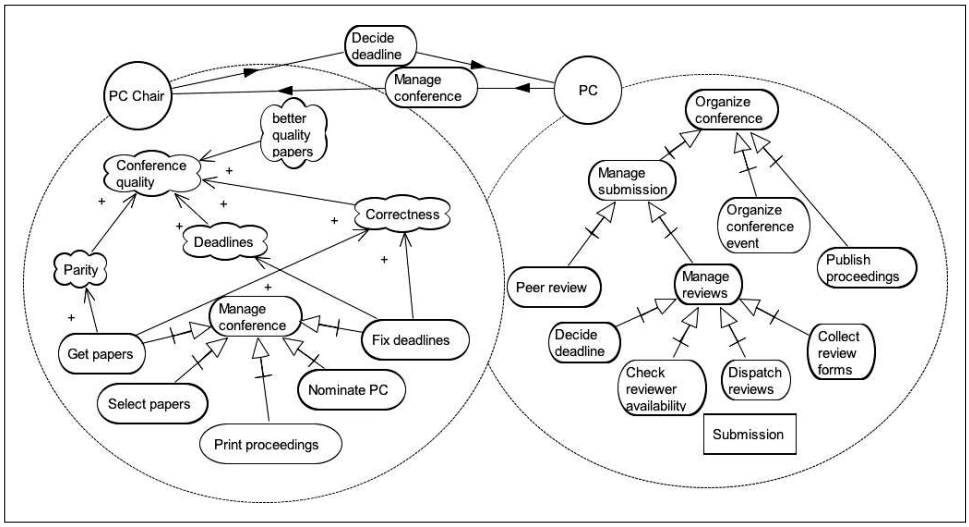


Figure 9 Early requirements of CMS: goal diagram



Additional analysis is used to decompose goals into sub-goals, which may include capturing alternative ways to achieve a given goal. These alternatives are captured by OR-decomposition. At this stage, non-functional requirements can be represented as soft-goals. It is often the case that choosing one alternative over another, leads to the achievement of different soft-goals. Thus, using OR-decomposition, it is possible to compare different alternatives and select the most appropriate one based on the desired soft-goals. This type of analysis can be used on goal diagrams such as those depicted in Figure 9. In this diagram, only two actors, PC and PC Chair, are represented. The diagram includes two goal dependencies, Manage conference and Decide deadlines. The goal Manage conference is analysed from the point of view of its responsible actor, PC Chair. Manage conference is AND-decomposed into several sub-goals: Get papers, Select papers, Print proceedings, Nominate PC and Decide deadlines. In addition, various soft-goals are specified inside the actor goal diagram along with their contribution relationships to/from other goals. For example, the soft-goal Better quality papers positively contributes to the soft-goal Conference quality.

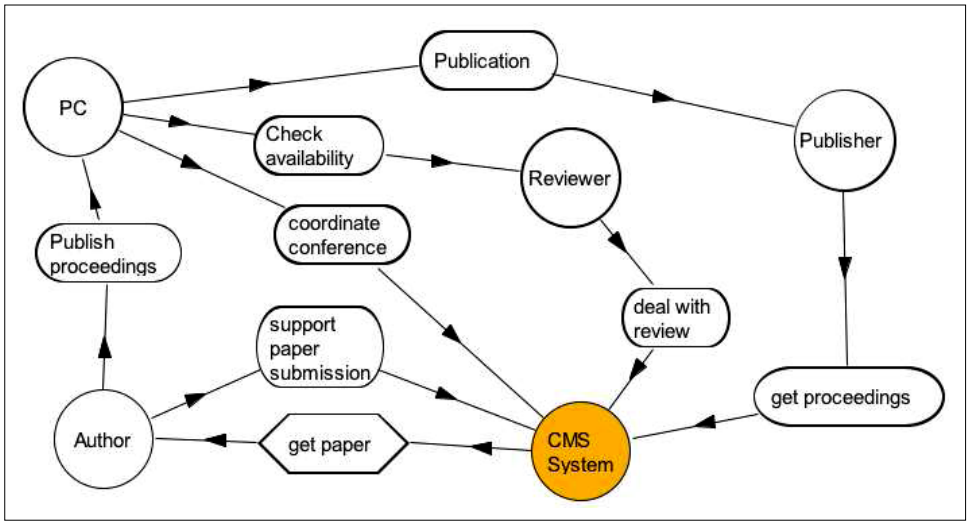
Goal diagrams are created and viewed dynamically in *TAOM4E*. Each actor in the model has a goal diagram, which can be dynamically opened and closed. These goal diagrams appear as balloons attached to the respective actors, which allows the analyst to dynamically visualise the internal perspective of each actor. Notice also that the tool supports the analyst in identifying the elements to be analysed. For instance, goals that have been delegated to an actor through dependency relationships appear automatically in the actor goal diagrams. For instance, in Figure 9 the goal Manage conference automatically appears in the PC Chair actor's goal model after being delegated from the PC.

3.1.2 Late requirements

The Late Requirements phase is intended to capture the changes in the domain caused by the introduction of the system-to-be and the actual properties of the system. The phase starts by introducing a new actor, the system-to-be, into the domain model.

A partial view of the resulting model is shown in Figure 10 where the CMS System actor has been introduced. In practice, the analyst creates a new diagram inside the project and adds the new actor, specifying that it has the property of being a 'system actor'. The tool can also be customised to show system actors with a different colour with respect to domain actor to facilitate model reading.

Figure 10 Late requirements: actor diagram (see online version for colours)



During Late Requirements, the driving analysis questions can be stated as: *What are the goals that can be assigned to the system-to-be?* and *Which dependencies can be redirected from domain actors to the system?*

In answering these questions, several existing dependencies may be redirected or new dependencies established between the domain actors and the new CMS System actor. For example, new goal dependencies, Coordinate conference and Manage proceedings, have been introduced in Figure 10.

These goals are then analysed from the system actor perspective, as shown in Figure 11. The goals Coordinate conference and Manage proceedings are decomposed in new sub-goals. Moreover, operative plans are specified and linked to system goals. These links indicate a means-ends relationship between the plan and the goal. For example, in Figure 11, a means-ends relationship exist between the Manage decision goal and the accept and reject plans.

During this phase, formal properties can be defined for any entity in the model using the Formal Tropos language (Fuxman *et al.*, 2004), according to the metamodel defined in Bertolini *et al.* (2006). Formal Tropos is a language based on Linear Temporal Logic (LTL) that allows specification of constraints, such as the requirement for temporal sequencing in the fulfilment of the goals *assign_papers_to_reviewers* and

collect_reviews (*i.e.*, the assignment of the papers to the reviewers must be achieved before the collection of the reviews). This constraint can be expressed in Formal Tropos as shown below.

Global assertion F(

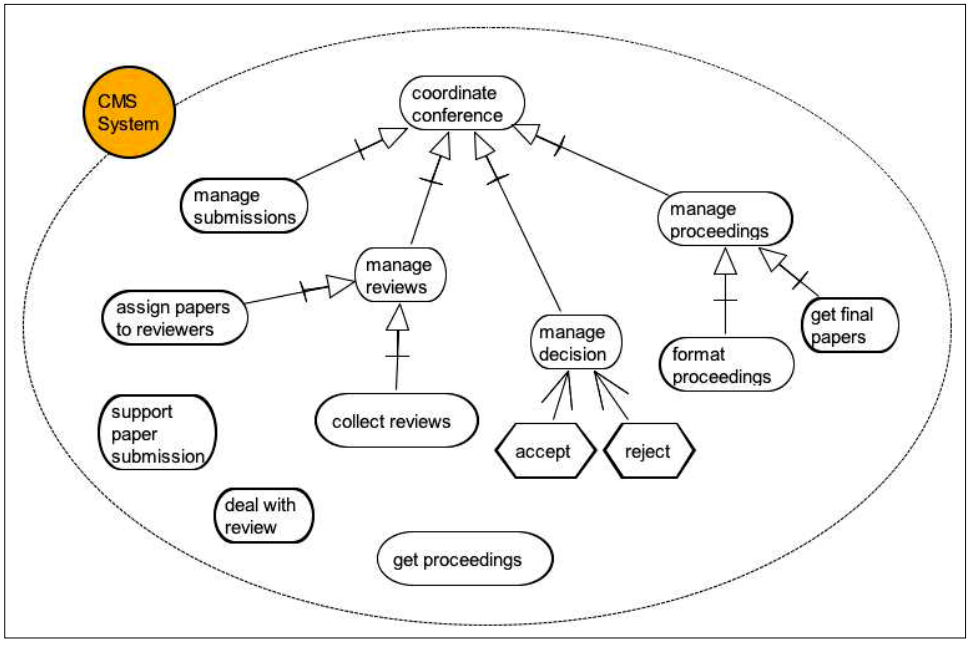
∀ cmss : CMS System

→ ∀ cr : collect_reviews (cr.**actor** = cmss

→ ∀ aptr : assign_paper_to_reviewers (aptr.**actor** = cmss → **Fulfilled**(aptr))))

The specification of the constraints is supported by *TAOM4E* via the annotation of the *Tropos* models and is an important feature that allows the designer to formally check the model using model checking techniques and tools, as described in (Perini and Susi, 2005).

Figure 11 Late requirements: goal diagram (see online version for colours)



The artefacts resulting from the Late Requirements phase are the extended domain model and the Late Requirements diagrams.

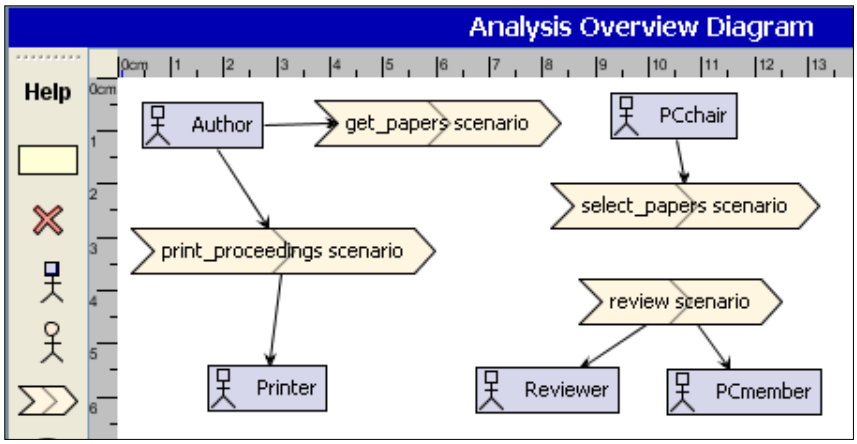
3.2 Prometheus and PDT

Prometheus assumes that the initial ideas for the system are captured – at least in a few paragraphs. During System Specification or Analysis this description must be elaborated and explored, in order to provide a sound basis for system design and development.

3.2.1 Analysis overview model

Typically, in Prometheus, the development of the System Specification begins with identifying the external entities[11] (referred to as *actors*) that will use or interact in some way with the system and the key *scenarios* around which interaction will occur. This is done in PDT using the 'Analysis Overview Diagram'. In Figure 12 we identify Author, Printer, PCchair, PCmember and Reviewer as the entities that will interact with the system. We associate the actors with the four scenarios that correspond to the main functionality of the system: get papers, review, select papers and print proceedings, indicating with which scenarios each actor will be associated. Thus reviewer and PC member are involved with the review scenario, while Author and Printer are involved with print-proceedings.

Figure 12 Initial analysis overview diagram (see online version for colours)



Having linked actors to scenarios, this diagram is then refined by identifying the *percepts* that are input to each scenario and the *actions* produced by the system for each scenario, linking them to the appropriate actors as shown in Figure 13. For example, an author submits a paper as a percept (input) to the system and the system performs an action of sending an acknowledgement back to the author. During definition of percepts and actions new links between actors and scenarios maybe established, as is the case here where a notification is sent from the select papers scenario to the Author, thus connecting the Author to this scenario as a recipient of the message (action). The analysis overview diagram thus defines the *interface* to the system in terms of the percepts (inputs) and actions (outputs).

3.2.2 Scenario model

The next step is to specify the details of the scenarios that we identified in the analysis overview diagram. A scenario is a sequence of *structured steps* where each step can be one of: *goal*, *action*, *percept* or *(sub)scenario*. Each step also allows the designer to indicate the *roles* associated with that step, the *data* accessed and a *description* of the step. These preliminary goals, roles and data that are identified are used to automatically propagate information into other aspects of the design. As steps are defined, the relevant entities are created if they do not yet exist. Figure 14 illustrates the steps of the paper

reviewing scenario where the first step is a goal to invite reviewers, associated with the Review_Management role and accesses the ReviewerDB (a data structure to store reviewer details, their preferences and paper assignments). This is followed by a goal step collect prefs to collect the preferences of the reviewers, a goal assign_reviewers to split the available papers between the reviewers, considering preferences, an action give_assignments to tell reviewers which papers they should review, a percept or input review report that comes into the system from each reviewer and a goal collect_reviews where all reviews for a particular paper are assembled.

Figure 13 Refined analysis overview diagram (see online version for colours)

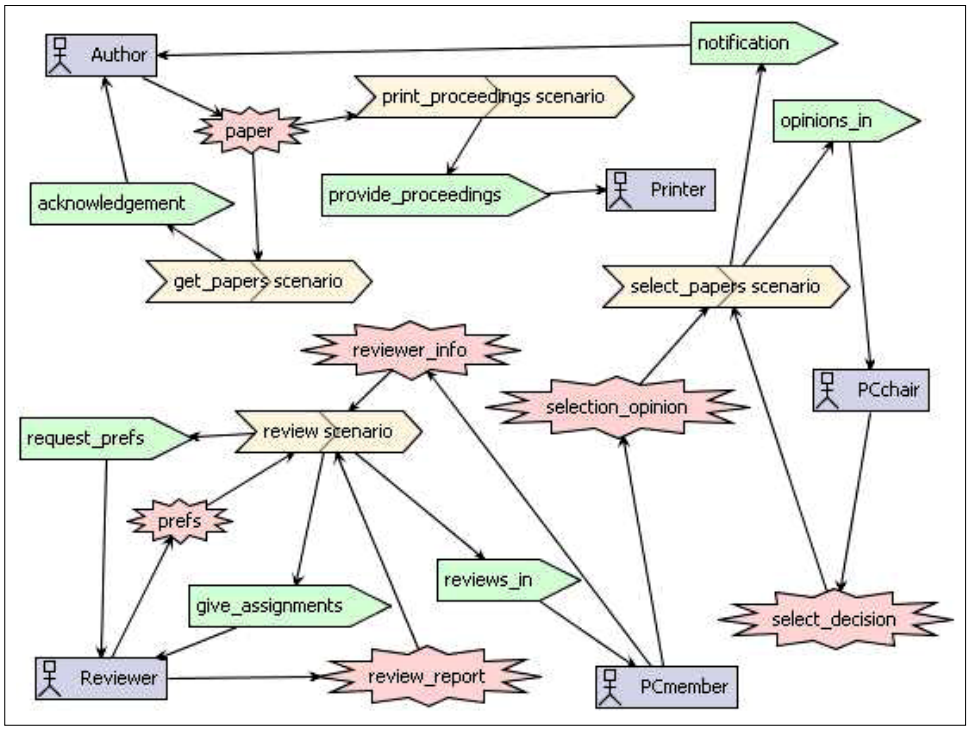
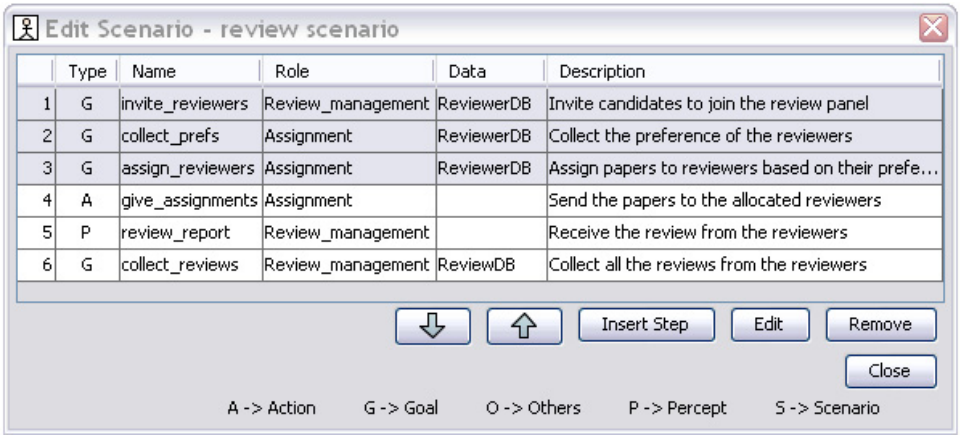


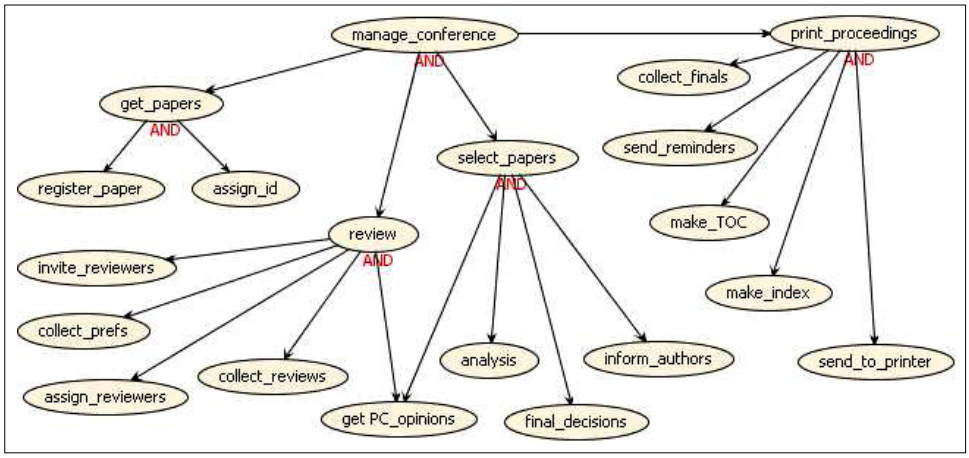
Figure 14 Scenario example – paper review (see online version for colours)



3.2.3 Goal model

By default, PDT creates a goal for every scenario, with the same name as the scenario. This is the goal that the scenario is intended to achieve. The name of the goal can be changed and, if desired, the same goal can be associated with multiple scenarios, although this is not usually the case at the most abstract level of the Analysis Overview diagram. The goals, created from the scenarios, are automatically placed into the 'Goal Overview Diagram', where goal hierarchies further describing the application are developed. For each goal, we identify its sub-goals by asking the question "how can we achieve this goal?". This can result in either a series of smaller subgoals, which are part of achieving the goal (AND decomposition) or in alternative approaches to achieving the goal (OR decomposition). The AND/OR is annotated below the parent goal, with the default being AND. Figure 15 shows the goals of the conference management system.

Figure 15 Goal overview diagram (see online version for colours)



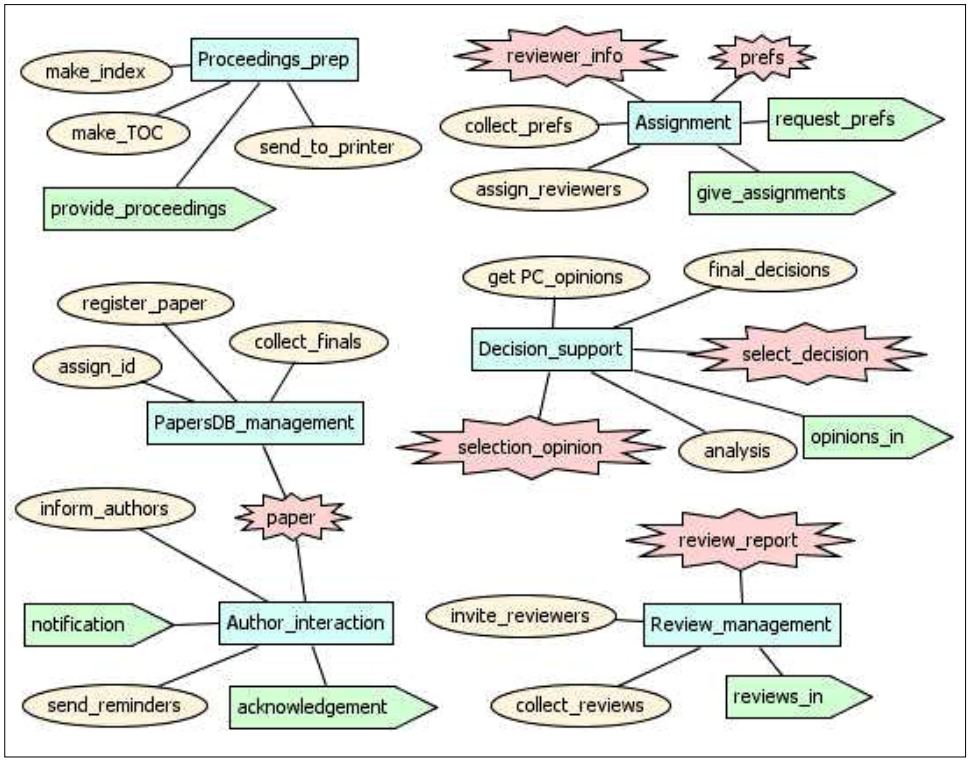
3.2.4 Role model

There is typically substantial iteration between scenario development and goal hierarchy development until the developer feels that the application is sufficiently described/ defined. At this stage, goals are grouped into cohesive units and assigned to *roles*, which are intended as relatively small and easily specified chunks of agent functionality. The percepts and actions are then also assigned to the roles appropriately to allow the roles to achieve their goals. This is done using the 'System Roles' diagram.

For example, Figure 16 shows that the 'Assignment' role is responsible for the goals to collect preferences (from the reviewers) and assign papers (to the reviewers). To achieve these goals, the role needs the input (reviewer_info) and reviewer preferences (prefs) and should perform the actions of requesting preferences from reviewers (request_prefs) and giving out the paper assignments (give_assignments).

Prometheus deliberately discusses only roles at this stage, leaving decisions about which agents the system should have until Architectural Design, when some analysis can be done on what is the preferred system structure.

Figure 16 System roles diagram (see online version for colours)



3.2.5 Results of analysis

The most important artefacts produced with PDT in this phase are the Scenario Model and the Role Model (incorporating the Goals), which are directly used in the Architectural Design. The Analysis Overview is also important as part of the conceptual design documentation for understanding the purpose of the system.

3.3 O-MaSE and aT^3

There are several Requirements and Analysis tasks used in O-MaSE and supported by *aT*^{3} that result in the creation of various requirements and analysis models. These models include the Goal Model, the Organisation Model, the Role Model, the Role Description Document and the Domain Model. Each model is demonstrated for the CMS example with the exception of the Role Description Document due to the simplicity of the CMS system and space constraints. The first step in O-MaSE is almost always to create the Goal Model. Once an initial version of the Goal Model has been defined, the Organisation Model is defined followed by the Role Model and Domain Model.

Figure 17 CMS AND/OR goal model (see online version for colours)

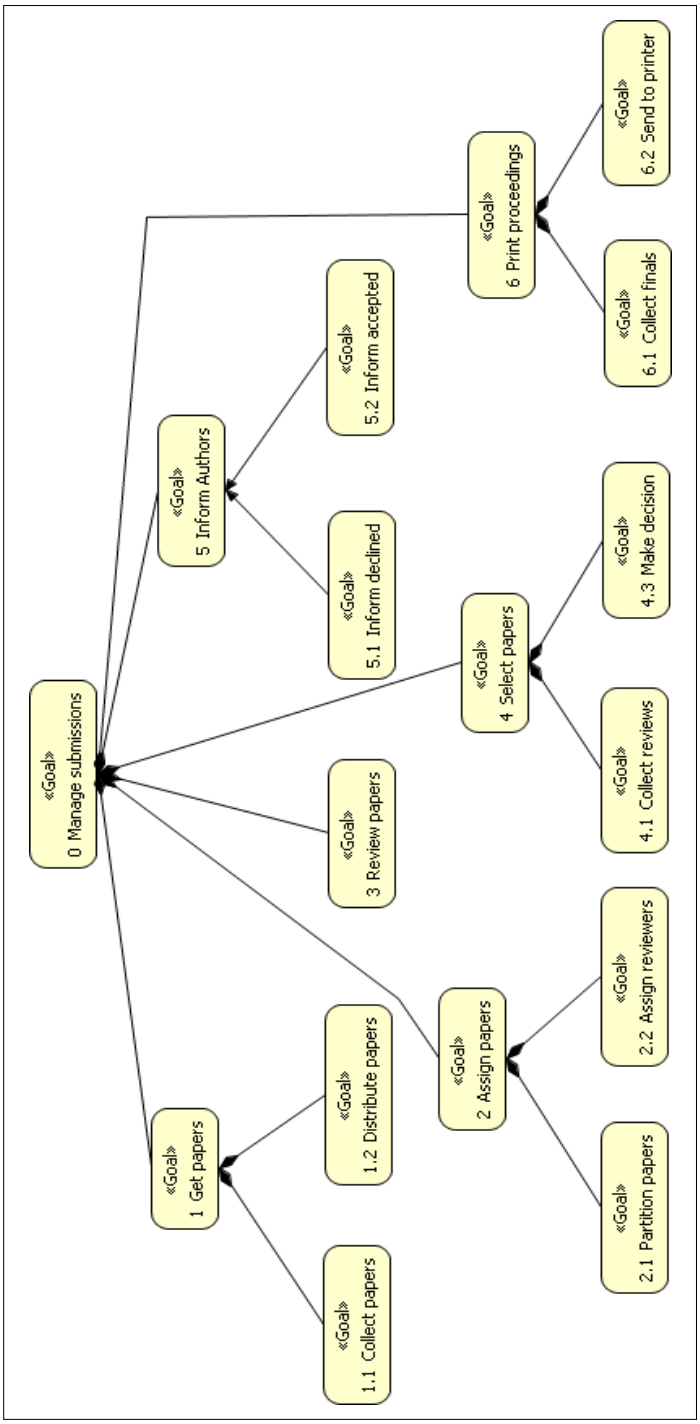
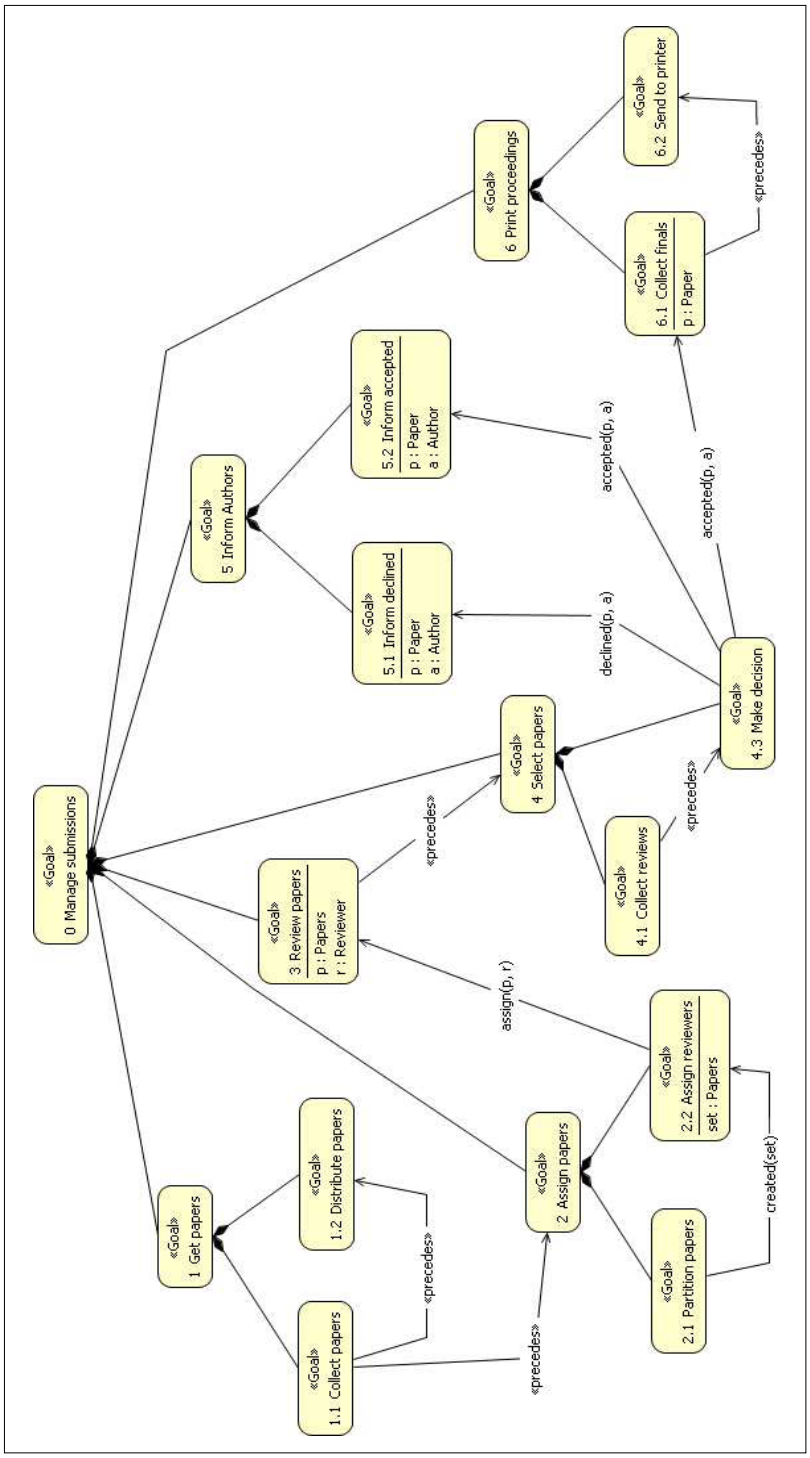


Figure 18 CMS GMoDS goal model (see online version for colours)



3.3.1 Goal model

The first step in an O-MaSE compliant process is to create an initial Goal Model that captures the essential requirements of the CMS system as defined in the system definition or requirements documents. The Goal Model is defined in aT^3 using the Goal Model diagram. The initial Goal Model for the CMS system is shown in Figure 17. The Model Goals task uses traditional AND/OR refinement (van Lamsweerde *et al.*, 1998) to decompose the top-level CMS goal, `Manage submissions`, into six AND-refined subgoals: `Get papers`, `Assign papers`, `Review papers`, `Select papers`, `Inform authors` and `Print proceedings`. An arrow with a filled diamond is used to represent AND-refinement while an arrow with a filled arrowhead is used to represent OR-refinement. The semantics of AND-refinement requires that all the children of a parent goal be achieved in order to achieve the parent, while in OR-refinement, achievement of only one of the children goals is required to achieve the parent. Each goal in the model is annotated by the designator ≪Goal≫. All the subgoals except `Review papers` are further decomposed into subgoals that define what must be accomplished in order to achieve the given goal. For instance, the `Select papers` goal is AND-refined into a `Collect reviews` goal and a `Make decision` goal. Notice that the `Inform authors` goal is OR-refined into an `Inform declined` and `Inform accepted` subgoals. Obviously, the subgoal used to satisfy the `Inform authors` goal is based on the decision made whether to accept or reject the paper. aT^3 ensures that the goal models drawn use consistent AND/OR refinement and form a valid tree.

The Goal Refinement task takes the initial Goal Model and adds additional information to capture the dynamism associated with the CMS system. Specifically, the initial model is refined into a model based on the Goal Model for Dynamic Systems (GMoDS) (Miller, 2007). GMoDS models are also captured in aT^3 via the Goal Model diagram, which is simply an extension of the goal model created previously. GMoDS introduces three concepts into AND/OR goal modelling approaches to handle goal sequencing and the creation of goal instances and parameterised goals. Sequencing of goals is provided by goal precedence, which specifies that one goal must be achieved before a second goal can be achieved. Goal instances are created based on events that occur during system operation. Goals without a specific trigger are created at system initialisation, while other goals are created when specific events occur. Finally, goals can be parameterised to fully define the purpose of the goal. For instance, in the CMS system, there is a goal to `Review papers`. However, this goal is ambiguous until the analyst specifies which set of papers to be reviewed. Thus, a parameter is added to the goal to specify the papers is to be reviewed. Again, aT^3 supports the modeller by automatically verifying that certain rules about circular precedence and triggering relations are not violated.

The GMoDS Goal Model for the CMS system is shown in Figure 18. The GMoDS model has the same basic shape as shown in Figure 17 but with additional arrows between goals showing precedence and goal triggering as well as parameters for several goals. In Figure 17, precedence between goals is shown by an arrow labelled with the ≪precedes≫ designator while triggers are represented by arrows between goals with an event name and a set of parameters in the form *event(p1,...pn)*. Reading Figure 18, it is clear that the `Collect papers` goal precedes both the `Distribute papers` and `Assign papers` goals. Thus, once the `Collect papers` goal has been achieved,

the papers may be distributed and the `Partition papers` goal (a sub-goal of `Assign papers`) can begin. The trigger between `Partition papers` and `Assign reviewers` denotes that each time a set of papers is *created* during the pursuit of the `Partition papers` goal, a new goal is instantiated for that set. Once the `Partition papers` goal is achieved, the pursuit of the `Assign reviewers` goal can begin on each of the `Assign reviewers` goals. As an assignment is made, the *assign(p, r)* trigger creates a new goal to `Review papers` for each paper set and reviewer assigned.

When all the `Review papers` goals have been achieved, the `Select papers` goal can be pursued via its subgoals: `Collect reviews` and `Make decision`. When the `Collect reviews` goal is achieved, then the `Make decision` goal can be pursued. As a decision is made on each paper, a *declined(p, a)* or *accepted(p, a)* event occurs. If a paper is declined, an `Inform decline` goal for that paper is instantiated while if a paper is accepted, both an `Inform accepted` and `Collect finals` goal is instantiated for that paper. Once all the `Collect finals` goals are achieved, then the `Send to printer` goal can be pursued. Assuming the `Inform authors` goals have been achieved, achievement of the `Send to printer` goal achieves all the sub-goals and the overall system goal is achieved.

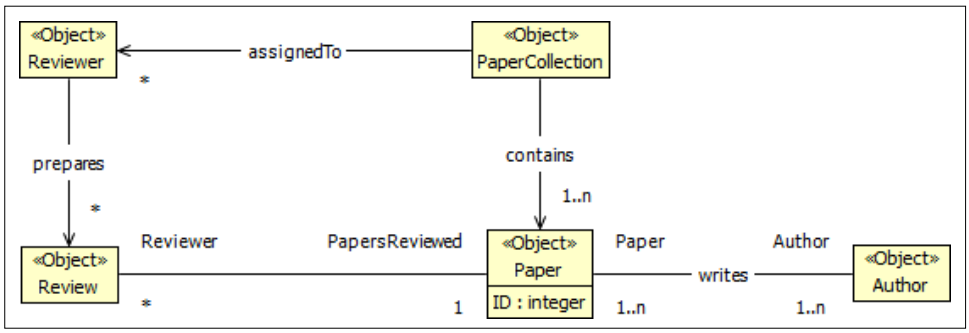
3.3.2 Domain model

The O-MaSE Model Domain task is used to create the Domain Model in aT^3, which is a very important model that is referenced by the Goal Model and extensively used in the Design Phase. The Domain Model defines the domain-specific entities that are referenced or manipulated by agents, plans and capabilities as shown later in Section 5.3. In the Goal Model, the goal and event parameters are defined as elements of the Domain Model. Thus, the Domain Model is usually developed in conjunction with the Goal Model and modified as needed during the Design Phase.

The Domain Model for the CMS system is shown in Figure 19. aT^3 supports development of the Domain Model by providing traditional object-oriented concepts of objects (which actually represent object classes – there are no object instances in a domain model), associations, inheritance and aggregation/composition. Each object (class) in the domain has two further descriptions that include attributes and constraints; these are shown in the `Paper` object in Figure 19. Association between objects (classes) can be unidirectional or directed, with traditional role names provided in order to reference an object from an associated object (class). The notion of association multiplicities are also provided to allow constraints to be added to the number of associations allowed.

In the CMS Domain Model, the `Reviewer` and `Author` objects will eventually be represented by an agent and an external actor respectively. As the development progresses through the process, the Domain Model will be updated to reflect these design decisions, since agents and actors are actually sub-types of domain objects. The actual objects – `Review`, `Paper` and `Papers` – define types that are used in the Design Phase. Specifically, they are used to define parameters of goals, messages and actions.

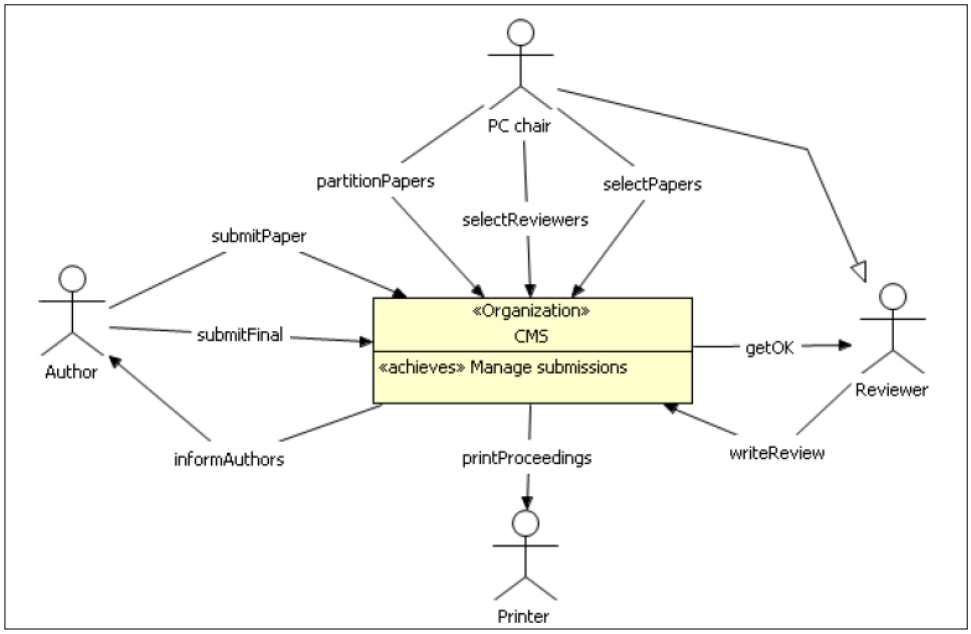
Figure 19 CMS domain model (see online version for colours)



3.3.3 *Organisation model*

The Organisation Model is created in aT^3 via the Model Organisation task, which takes as input the GMoDS Goal Model derived previously. The aim of the Model Organisation task is to identify the system's (which is referred to as the organisation) interfaces with external actors. In the case of the CMS system (see Figure 20), the system interfaces with the committee (including the `PC chair` and the `reviewers`), the `Authors` and the `Printer`. The various ways that the actors interact with the system are modelled as protocols, which are represented by arrows from the initiator of the protocol to the responder. The initiator and responder of a protocol must be either an external actor or the organisation. The system is represented as an organisation, which is denoted using the ≪Organization≫ designator.

Figure 20 CMS organisation model (see online version for colours)



As stated above, the CMS organisation interacts with Authors, the PC chair, Reviewers and the Printer. Each of these are shown as actors in Figure 20. Using the system description, the protocols required for interaction between the organisation and the actors are identified. In the CMS system, an Author submits papers to the system using the submitPaper protocol. After being reviewed, the CMS notifies the Author whether their paper is accepted or rejected via the informAuthor protocol. If the paper was accepted, the Author then submits the final version of the paper using the submitFinal protocol. The PC chair actor works with the CMS by partitioning papers into sets via the partitionPaper protocol and then assigns various reviewers to review those sets of papers via the selectReviewers protocol. Once the reviews are complete, the PC chair makes the final selections via the selectPapers protocols. The Reviewers accept or reject their assignments via the getOK protocol and submit their reviews via the submitReviews protocol. Finally, the final papers are sent to the Printer for printing via the printProceedings protocol.

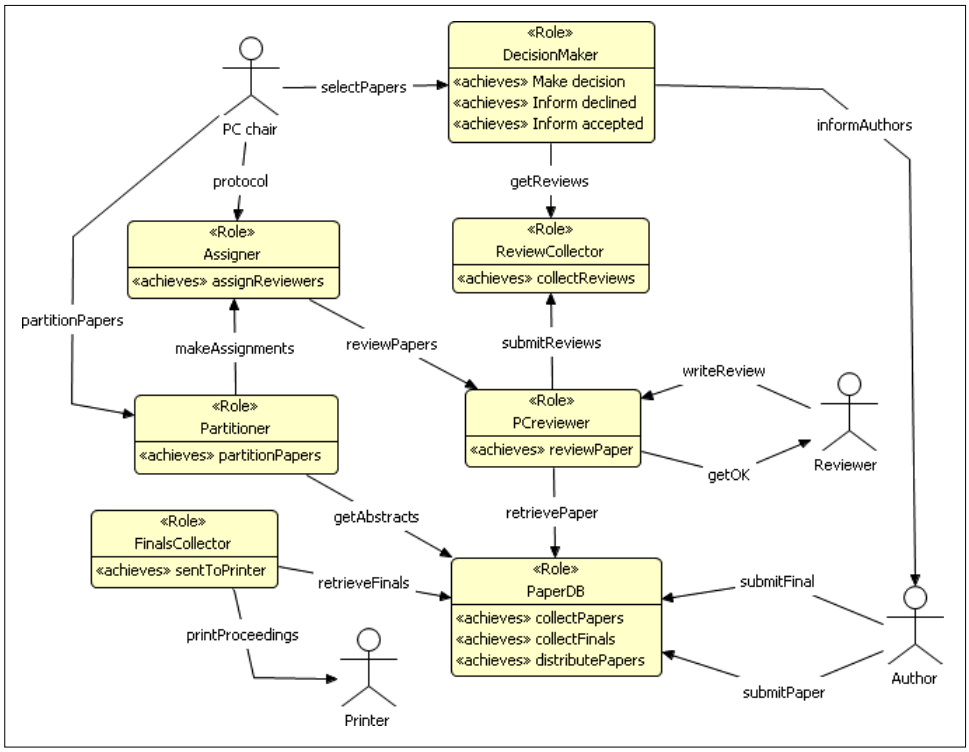
3.3.4 Role model

Once the Organisation Model and Goal Model have been sufficiently defined, the task Model Roles is used to create a Role Model. This focus of the Model Roles task is to identify the roles required internal to the organisation and their interactions (defined via protocols) with each other. The actors from the Organisation Model should show up as actors in the Role Model and the protocols between the actors and the organisation must be mapped to protocols between those actors and specific roles in the system. Thus, the Role Model is a refinement of the Organisation Model. In addition, each leaf goal in the GMoDS Goal Model must be assigned to a role in the Role Model that can achieve it, as denoted by the ≪achieves≫ designator in the body of the role. Thus, each role should achieve at least one leaf goal although, in general, a role may achieve multiple leaf goals. aT^3 supports these refinement rules by automatically providing warnings to the user if the rules are violated. For instance, if a leaf goal in the Goal Model is not assigned to a role via an ≪achieves≫ relation, a warning will be displayed in the Eclipse Problem window stating that the leaf goal has not been assigned.

The Role Model for the CMS system is shown in Figure 21. In the CMS system, there are seven roles: the PaperDB, the Partitioner, the Assigner, the PCreviewer, the ReviewCollector, FinalsCollector and DecisionMaker. The PaperDB role acts as the collection and distribution mechanism in the CMS. Authors (where the Author actor represents only a single contact author of a paper) submit papers to the PaperDB, while the Partitioner, PCreviewer and Finals Collector roles access the papers, abstracts and final versions via protocols with the PaperDB. When all the papers have been submitted, the PC chair interacts with the Partitioner role to look at the various abstracts and assign them to groups to be assigned reviewers. Once this task is complete, the PC chair interacts with the Assigner role to select reviewers to assign to each set of papers. The Assigner role then interacts with the PCreviewer role via the reviewPapers protocol, which interacts with the Reviewer via the getOK protocol. The Reviewer then reviews the papers and submits them to the PCreviewer role using the writeReviews protocol. The PCreviewer role then sends the reviews to the ReviewCollector role. Once all the reviews have been submitted, the PC chair interacts with the DecisionMaker role to select papers for the conference. The

status of the papers are relayed to their authors by the `DecisionMaker` role via the `informAuthors` protocol. Once the `Author` completes the final version, the paper is submitted to the `PaperDB` via the `submitFinal` protocol. When all the final papers have been submitted, the papers are then forwarded to the `Printer` from the `FinalsCollector` via the `printProceedings` protocol.

Figure 21 CMS role model (see online version for colours)



The completion of the Goal Model, Domain Model, Organisation Model and Role Model concludes the analysis phase of this O-MaSE compliant process. Each model in the analysis phase is supported by an associated aT^3 diagram type. During the analysis phase, aT^3 ensures consistency between the models via its validation engine. When completed, the models describe what the system must do and the logical elements required for the system to achieve its overall goals.

3.4 Discussion

The analysis phase covers the activities of both domain analysis and system specification. All three toolkits and their associated methodologies do support both of these aspects. However, the emphasis is different. In *TAOM4E*, domain analysis takes on a central role and the majority of the tool support is oriented towards this. In PDT and aT^3 the development of a detailed understanding of the system to be built is the more central of the two aspects.

All three toolkits use the concepts of actor and goal at this stage, although the semantics are slightly different. *TAOM4E* considers actors to be the humans that are important in the domain and then later introduces the system-to-be as an additional actor. PDT and aT^3 both consider actors to be the humans or software systems that will interact with the system-to-be. In *TAOM4E* goals at this stage are the goals of the actors, while in PDT and aT^3 they are the goals of the system.

All three tools exploit AND/OR decomposition of goals into more detailed goals. aT^3 complements the information of a typical AND/OR goal model to allow representation of an expanded set of constraints between the goals. In particular, aT^3's GMoDS diagrams allows the specification of temporal sequences of goals and constraints related to the instantiation of new goal instances. These constraints are similar to those allowable in *TAOM4E* via entity properties in Formal *TAOM4E* (Fuxman *et al.*, 2004). While Formal *TAOM4E* is more expressive, the GMoDS model is much simpler and easier to use in the design process. In PDT temporal relationships between goals are partially captured by the sequencing of goals within the steps of a scenario.

Table 4 summarises the concepts used within each of the tools at the Analysis stage.

Table 4 Concepts in the analysis stage of the three tools

Taom	*PDT*	aT^3
Actor, goal, soft-goal	Actor, goal	Actor, goal
Resource, plan	Role, scenario	Role, protocol
	Percept, action, data	

Both PDT and aT^3 use roles, although in slightly different ways. In both PDT and aT^3, roles capture aspects of the system and are associated directly with system goals. However, in PDT, roles are also associated with percepts and actions (Figure 16), while in aT^3, roles are associated with required capabilities (Figure 21), which are left until a later stage in PDT.

Protocols are introduced in aT^3 at the analysis stage to capture interaction between actors and the system (Figure 20). A protocol defines the possible legal sequences of steps in a particular interaction. At this stage in aT^3, the protocols are identified but not specified in detail. Within PDT, scenarios (Figure 14) play a somewhat similar role in that they also capture interactions between the system and actors in terms of percepts (received by the system) and actions (produced by the system). Unlike a protocol that defines all possible legal sequences of steps, a scenario captures one particular sequence. Both aT^3 and PDT leave the detailed protocol definitions for later in the Architectural design stage.

Prometheus is the only methodology, hence PDT the only tool, that focuses on identification of percepts as the input to the agent system and actions as the output (see PDT Figure 13). While percepts and actions are considered in O-MaSE(aT^3), they are embedded in plans and capabilities defined later in the detailed design.

TAOM4E, with its greater emphasis on domain analysis includes dependencies between actors and goals at this stage (Figure 8). Actors form dependencies based on goals or resources. In the late requirements goal diagrams in *TAOM4E* (Figure 11) plans are also identified (but not detailed) which operationalise the goals. Both PDT and aT^3 leave this to the Detailed design stage.

The notion of soft-goals (*e.g.*, in Figure 9, ensure only high quality papers are accepted) is explicit in *TAOM4E*, while in PDT goals can be annotated as soft-goals by using the notes feature. It is, However, absent in aT^3.

As noted earlier the Analysis stage includes both domain analysis and system specification. Table 5 summarises the models related to the Analysis stage within each of the toolkits.

Table 5 Models in the analysis phase of the three tools

Taom	*PDT*	aT^3
Early requirements actor diagram	Analysis overview	Domain model
Early requirements goal diagram	Scenarios	Organisation model
Late requirements actor diagram	Goal overview	Goal model
Late requirements goal diagram	System roles	Role model

TAOM4E, with its focus on the Early Requirements phase includes modelling and analysis of the domain without considering the system-to-be. The Early Requirements Actor and Goal diagrams of *TAOM4E* used for this purpose are substantially richer and more complex than the Analysis Overview model of PDT or the Domain Model of aT^3.

PDT with Scenarios and aT^3 with Protocols do some preliminary modelling of the system dynamics at this stage, which is not covered by *TAOM4E*. PDT and aT^3 also model roles as discussed earlier in this section via the System Roles Diagram and the Roles Model respectively.

4 Architectural design

Following the system analysis and specification phase, the architectural design establishes the structure of the system being developed. Each of the systems differ slightly in what exactly they cover in this phase, and where they draw the line between architectural and detailed design. We continue with our description of each of the systems in turn.

4.1 Tropos and TAOM4E

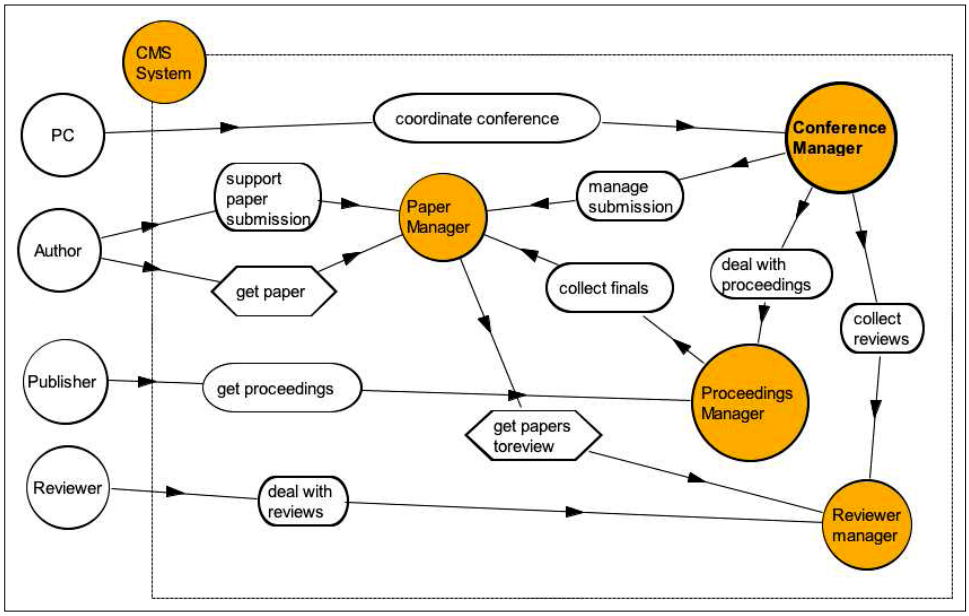
The *Tropos* Late Requirements model is the basis for the definition of the system architecture. The *Architectural Design* artefact defines the system's overall structure; it is represented in terms of its sub-systems and of their interdependencies. In the multi-agent paradigm, sub-systems are agents that can act independently and communicate with others through message passing. In order to build the architectural design, the engineer refines the system actor by introducing sub-actors, which are responsible for actually carrying out specific system goals. Criteria that guide the identification of the high level system goals to be assigned to the sub-actors include goals that have no relations between them (independent goals) and existing design patterns (Kolp *et al.*, 2003).

During this refinement activity, the engineer is faced with alternative decompositions. As in traditional software engineering approaches, designs that results in sub-systems with stronger internal cohesion and lower coupling should be selected.

The engineer creates an *Architectural Design* diagram in *TAOM4E* for each system actor defined in the *Late Requirements Analysis* phase. In this diagram, a dashed box associated to the system actor represents the system. Within the box, new actors, roles, and system agents may be created. Subsequently, a single goal, the whole goal tree or parts of it can be delegated from the system to the new actors, roles or system agents.

Figure 22 displays the resulting architectural design diagram for the CMS System actor. Based on the actor's goal model as shown in Figure 11, the engineer has decomposed the system into sub-actors. In our example the engineer has introduced four new actors. The Conference Manager manages the top-level goal coordinate conference, which was delegated to the system by the programme committee actor PC. The Paper Manager is delegated the goal support paper submission from the domain actor Author. In addition, some system actors depend on the Paper Manager to manage papers and depend on the domain actor Author actors to get papers. Similarly, the additional goals have been delegated to the Review Manager and Proceedings Manager actors as well. In our case study, the new sub-actors are agents in the CMS system.

Figure 22 Architectural design: CMS system decomposition into sub-actors (see online version for colours)



4.2 Prometheus and PDT

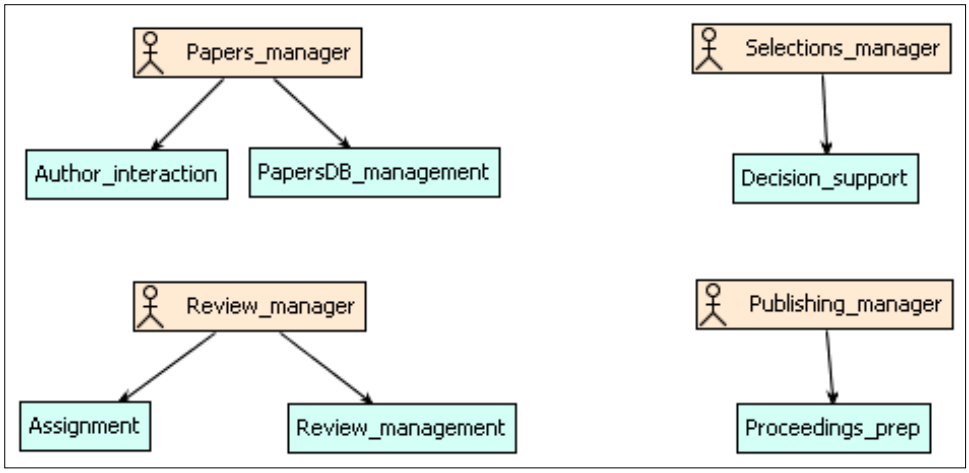
In Prometheus, the main tasks of Architectural Design are to decide the agent types (as collections of roles) and to define the agent conversations (protocols) that will happen in order to realise the specified goals and scenarios.

4.2.1 Agent types

PDT supports the process of making decisions about which agent types to have in the system, by providing a data coupling and agent acquaintance diagram to allow the developer to visualise data and communication coupling, which can influence decisions about which roles to group. Once decisions have been made about which roles to group into agents, this is captured in the 'Agent-Role Grouping Diagram'. Various properties and relationships are then propagated by the tool, from the roles to the agents.

Figure 23 shows the roles of assigning papers to reviewers (Assignment) and managing the review process (review_management) as being part of a Review_manager agent.

Figure 23 Agent-role grouping diagram (see online version for colours)



Once decisions have been made about how roles are grouped into agents, the information propagated from roles allows PDT to automatically place agents, percepts and actions onto the 'System Overview Diagram', with percepts and actions connected to appropriate agents. What must be done to complete this overview is to define interactions between the agents (protocols) and to add information (in the form of an icon, with associated descriptor) about any data shared between agents. When completed, this diagram provides an overview of the internal system architecture.

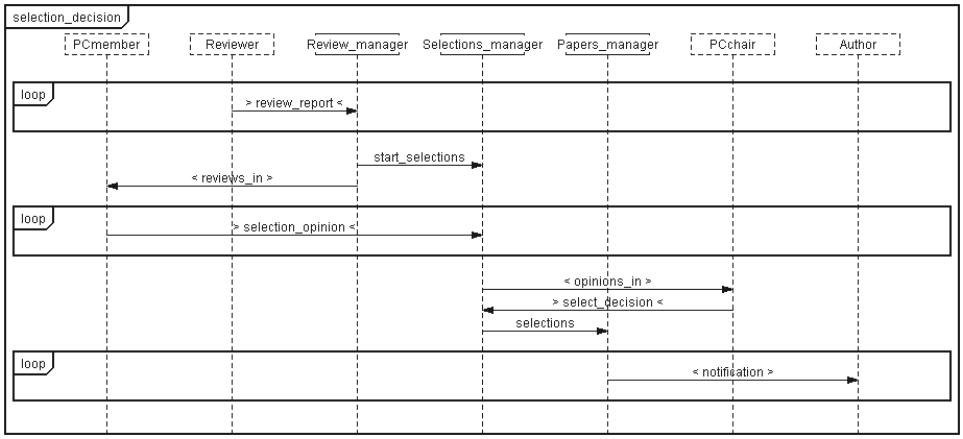
4.2.2 Protocols

Protocols can be instantiated by placing graphical icons onto the system overview diagram and then specifying their structure in the protocol specification window (available from the Entities menu or by double clicking the icon in the diagram). The structure of message flows is specified using a textual notation for describing a modified AUML2 (Winikoff, 2007) protocol specification. This can then be displayed (by selecting a tab on the pop-up window) as a figure which is similar in style to AUML2. Any messages (or other entities) specified in the protocol, but not yet existing in the design, are created automatically by PDT. Links between agents and protocol symbols are created automatically in the system overview diagram, based on the specification.

Prometheus modification of AUML2 allows percepts, actions and actors to be part of the protocol specification in PDT, in addition to messages and agents. This often provides a better understanding of a conversation structure than showing only messages between agents.

Figure 24 shows the AUML2-like diagram of the 'selection_decision' protocol, where interactions involve three agents and four actors (identified by the dotted squares in the diagram). Percepts (which always originate with an actor and go to an agent) are written as '>percept_name<' and actions (from an agent to an actor) are written as '<action_name>'.

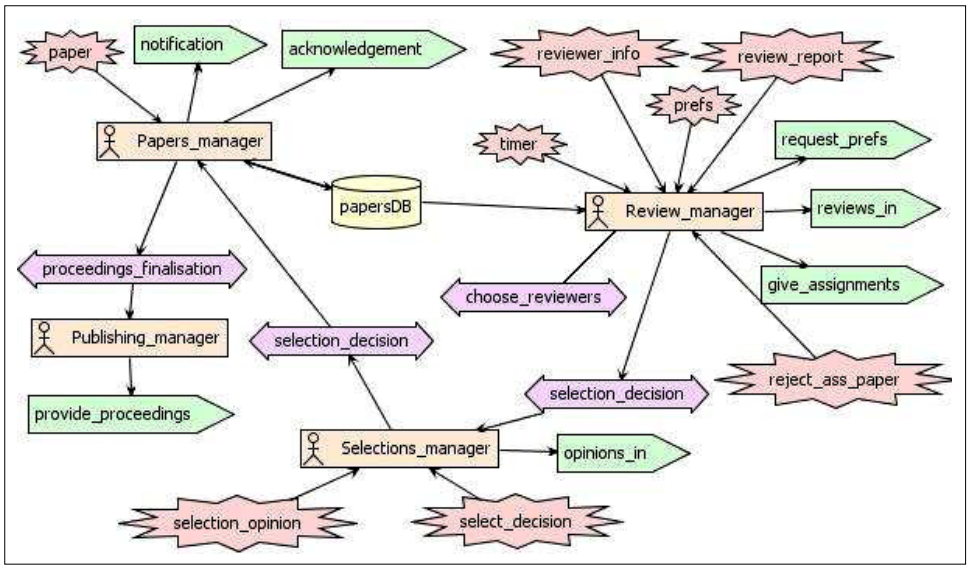
Figure 24 Selection decision protocol diagram



In Figure 24 showing the selection decision interaction, we see a loop with review reports arriving as percepts from the Reviewer actor(s) to the Review_manager agent. This is then followed by a message start_selections from the Review_manager agent to the Selections_manager agent when all reviews are in, as well as an external message reviews_in to the PCmember actor. There is then another loop where the PCMember actor(s) provide selection_opinion percepts to the Selection_manager agent, which, when these are all in, sends an external message (or action) opinions_in to the PCChair actor, who sends a select_decision percept regarding all the papers to the Selections_manager agent. This agent then sends a selections message to the Papers_manager agent who, in turn, sends external notification messages (actions) to the author actor(s).

Because conversations, or protocols, do include external actors, it is possible to have a protocol connected to only one agent. An example of this in Figure 25 is the 'choose_reviewers' protocol where the review manager interacts with reviewers to give out assignments. This protocol includes the review manager requesting preferences ('request_prefs' action) from the reviewers,[12] receiving the preferences ('prefs'percept), giving out assignments and so on.

Figure 25 System overview diagram (see online version for colours)



4.2.3 *System overview*

Figure 25 shows the system overview for our conference management system design. This diagram provides an overview of the internal system architecture in terms of the agents, their interactions with each other, the inputs (percepts) to and outputs (actions) from each agent and any shared data. For example, observing the 'Papers_manager' agent, we can see that it receives papers ('paper' percept) from authors (that are external to the system) and provides an acknowledgement (action) to them. It interacts with the 'Selections manager' agent via the 'selection_decision' protocol to be able to send authors a notification of accept/reject (action). It also interacts with the 'Publishing_manager' agent via the 'proceedings_finalisation' protocol to provide final versions of papers to publish the proceedings. The 'papersDB' is shared between the 'Papers_manager' and the 'review_manager'.

4.3 *O-MaSE and aT*^{3}

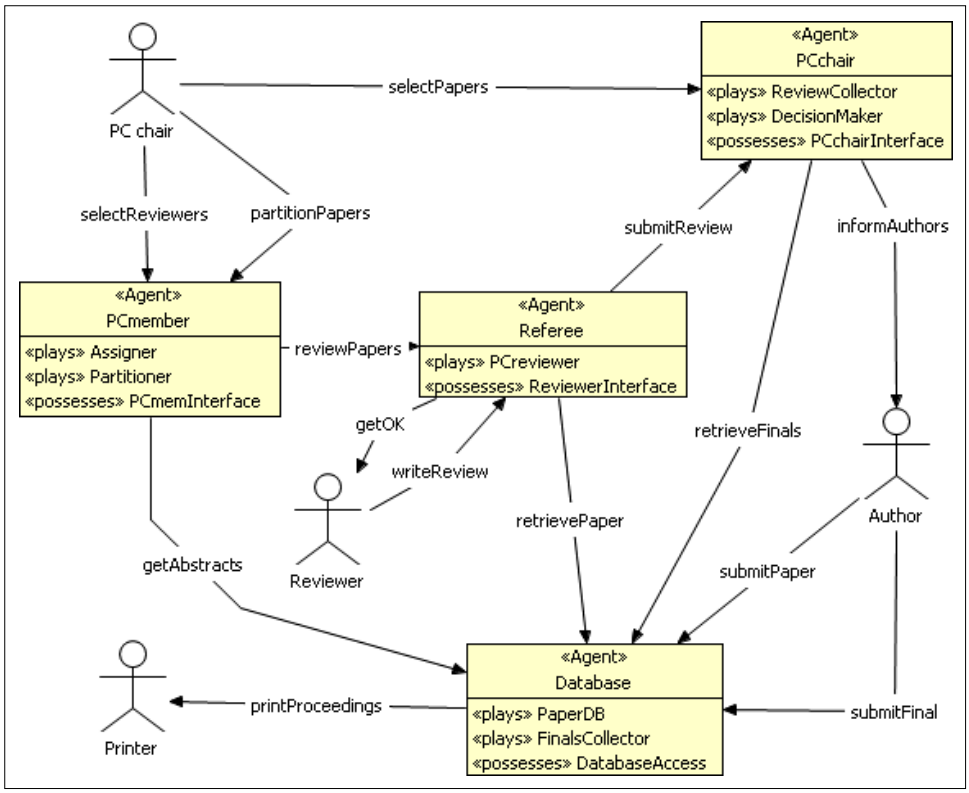
Once the analyst has defined what the system must do (via the Goal Model) along with the logical elements of the system (via the Domain, Organisation and Role models), the designer defines the architecture of the system. Essentially, the architecture of the system is defined by a set of agent types and a set of protocols between these agents, as is the case with Prometheus. The internal definition of the agent behaviour is left for the detailed design phase. To capture the types of agents in the system, the Model Agent Classes task is performed. After the designer has modelled all the agent classes and mapped the protocols from the Role Model to the Agent Class Model, the details of the protocols are defined via the Model Protocols task. Each of these models are supported by *aT*^{3} diagrams. In addition, *aT*^{3} provides a validation capability to ensure that the models are consistent as well. For instance, the designer may specify via an Agent Class

Model that a specific agent class can play a specific system role. The aT^3 validation component ensures that any such specification is consistent with the Role Model defined in the analysis phase.

4.3.1 Agent class model

The goal of the Model Agent Classes task is to translate the role model, which captures basic system functionality, into a form more amenable to implementation. In short, this means mapping roles to agent classes, which is captured in aT^3 Agent Class Model. The result of this mapping for the CMS system is shown in Figure 26. The roles that each agent has been assigned to play are embedded in the body of the agent classes and are prefixed with the designator ≪plays≫. The agent classes are denoted by the ≪Agent≫ designator.

Figure 26 CMS agent model (see online version for colours)



While the assignment of roles to agents is made by the designer, typical software engineering concepts such as coupling and cohesion should be used to evaluate the assignment. In the CMS system, two agent classes play two roles, while the other two classes play a single role each. The `PCmember` agent has been assigned to play both the `Assigner` and `Partitioner` roles and thus interacts with the `PC chair` actor. Likewise, the `PCchair` agent also plays two roles – `ReviewCollector` and

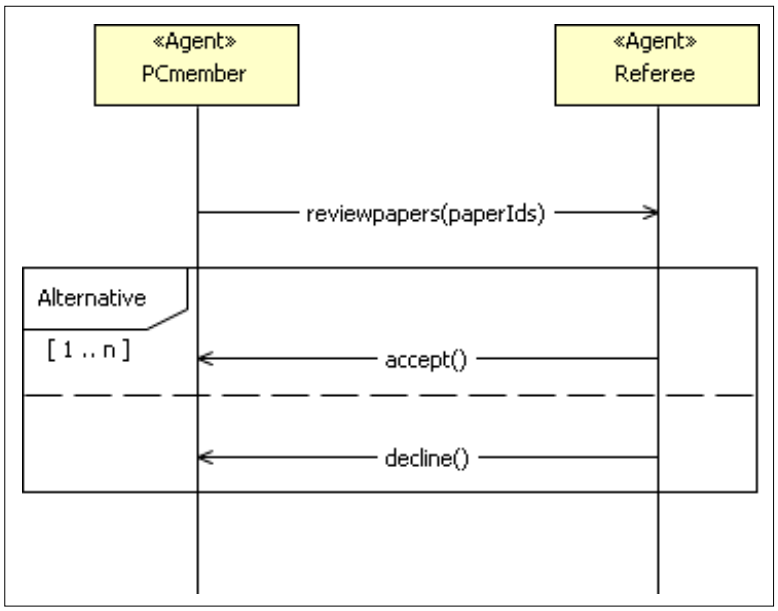
`DecisionMaker` – while also interacting with the `PC chair` actor. The `Referee` agent plays the `PCreviewer` role and interacts with the `Reviewer`, while the `Database` agent plays the `PaperDB` role and interacts with the `Authors` and the `Printer`. Notice that the protocols between roles in the Role Model have been mapped to protocols between the appropriate agents in the Agent Class Model.

After the Agent Model is complete, the agent classes and protocols have been identified but not defined. The next step in the architectural design is to model the details of the interactions between agent classes and between agent classes and external actors via the Model Protocol task.

4.3.2 Protocol models

The goal of the Model Protocols task is to define the details of the protocols identified in the Role Model and Agent Class Model. aT^3 supports defining these details via the aT^3 Protocol Model, which defines the protocol in terms of messages passed between agents or between agents and external actors using the AUML protocol notation in a fashion similar to Prometheus. As there were 13 protocols identified in the Agent Class Model (Figure 26), each of the 13 protocols must be defined in individual Protocol Models. The protocols are modelled using the AUML Interaction Diagrams (Huget and Odell, 2004), which allow designers to specify message sequences, alternatives, loops and references to other protocols.

Figure 27 CMS reviewPapers protocol model (see online version for colours)



Due to space constraints, only 3 of the 13 protocol models are presented here: `reviewPapers`, `submitReviews` and `retrievePapers`. Figure 27 shows the `reviewPapers` protocol, which defines the interaction between the `PCmember` and `Referee` agents, which are specified by the ≪Agent≫ designator (protocols can also

be specified between agents and actors using the same method). This protocol is very simple. The `PCmember` sends a `reviewpapers` message with a list of *paperIDs* for the `Referee` to review. The `Referee` may respond by either accepting or declining to review the set using the `accept` and `decline` messages respectively.

Figure 28 shows the `submitReviews` protocol, which defines the interaction between the `Referee` agent and the `PCchair` agent. In this protocol, the `Referee` sends several reviews via a `submit` message to the `PCchair` agent followed by a done message. There is no response by the `PCchair` agent.

Figure 28 CMS submitReviews protocol model (see online version for colours)

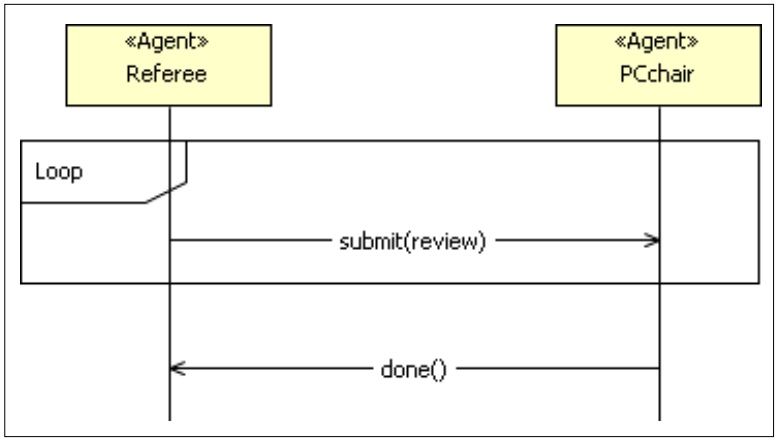


Figure 29 CMS retrievePapers protocol model (see online version for colours)

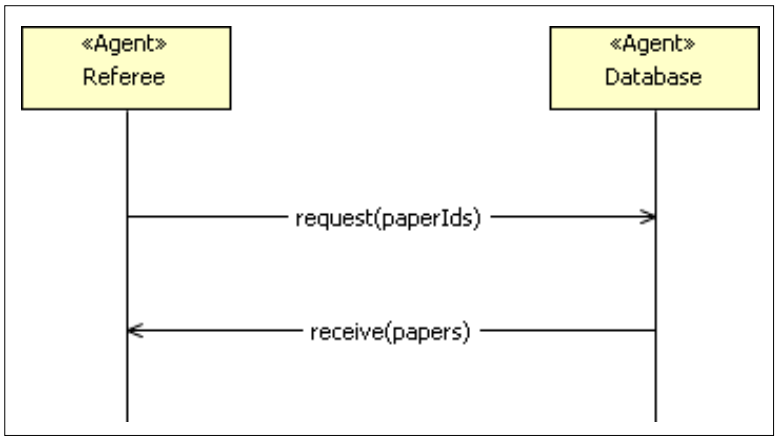


Figure 29 shows the `retrievePapers` protocol, which defines a simple request protocol between the `Referee` and `Database` agents. According to the protocol, the `Referee` issues a request to the `Database` for a set of papers via a `request` message. The `Database` simply responds with the appropriate set of papers in a `receive` message.

4.4 Discussion

The goal of Architectural Design is to capture the overall structure of the system. In this phase, we see two distinct approaches. *TAOM4E* focuses on refining the system actor into new actors, roles or system agents and then defining the internal goals and plans of those actors/roles/agents. In both PDT and aT^3, system roles were identified in the Analysis phase and thus Architectural Design is focused on assigning roles to agents and defining the interactions between those agents.

Table 6 summarises the concepts used within each of the methodologies at the Architectural Design stage.

Table 6 Concepts in the architectural design stage of the three tools

TAOM4E	*PDT*	aT^3
Actor, role, plan, goal	Agent, role	Actor, role
	Protocol, message	Protocol, message
	Action, percept, data	

All three tools (and corresponding methodologies) talk about both agents and roles during the Architectural Design phase. For *TAOM4E* these two terms are interchangeble and are also synonomous with system actor, which is the graphical concept icon used in the diagram. For both PDT and aT^3, roles are used to capture some limited aspect of required behaviour and are then grouped together to define the behaviour of the agents in the system.

Goals are used explicitly in *TAOM4E* to further refine the actors/roles/agents identified in this phase. (Although not shown in this example, aT^3 also allows roles to be further refined with an internal goal model.) PDT would expect further goal refinement to happen by revisiting the goal hierarchies of the initial phase or, more likely, to wait until the development of plans in the detailed design. In both PDT and aT^3, system level goals are associated with the roles defined in the Analysis phase. To show how to achieve internal goals, *TAOM4E* defines a set of plans for each goal identified; PDT and aT^3 define similar plans later in the Detailed Design phase.

Because *TAOM4E* does not explicitly define the agent interactions, only PDT and aT^3 use conversations or protocols. In both approaches, the protocols are used to define the allowable sequences of messages passed between system agents. Both also represent interactions with things outside the system as part of the protocols, with PDT using percepts and actions in protocol diagrams to denote these types of interactions. Finally, aT^3 attaches capabilities to each agent that can be used to define plans or percepts and actions in the Detailed Design phase.

Table 7 summarises the set of models related to the three methodologies during the Architectural Design phase.

Table 7 Models in the Architectural design phase of the three tools

TAOM4E	*PDT*	aT^3
Architectural design diagrams	Agent role grouping	Agent model
	System overview	Protocol model
	Protocol diagram	

All three methodologies and tools have a central model, represented graphically, that is the outcome of the Architectural design phase. For *TAOM4E*, this is the Architectural Design diagram, which decomposes the system actor into actors/roles/agents and relates them to goals and plans. In PDT, the central model and diagram is the System Overview diagram, capturing agents, their interactions with each other, shared data stores/structures and the interface of the system with the external environment. For aT^3, the Agent Model captures the assignment of roles to agents as well as the identification of interaction protocols between agents and between agents and external actors. Both aT^3 and PDT have additional models for specifying the details of protocols, both using AUML2. PDT has some subsidiary models that can be used as part of the methodology process (*e.g.*, the data coupling diagram which shows relations between roles and data, and the agent role grouping, which simply clusters roles into agents). *TAOM4E* includes some of the internals of agents at this level, whereas PDT and aT^3 both leave agent internals until detailed design. We will now explore how each of the tools support continuation of the design and development process.

5 Detailed design

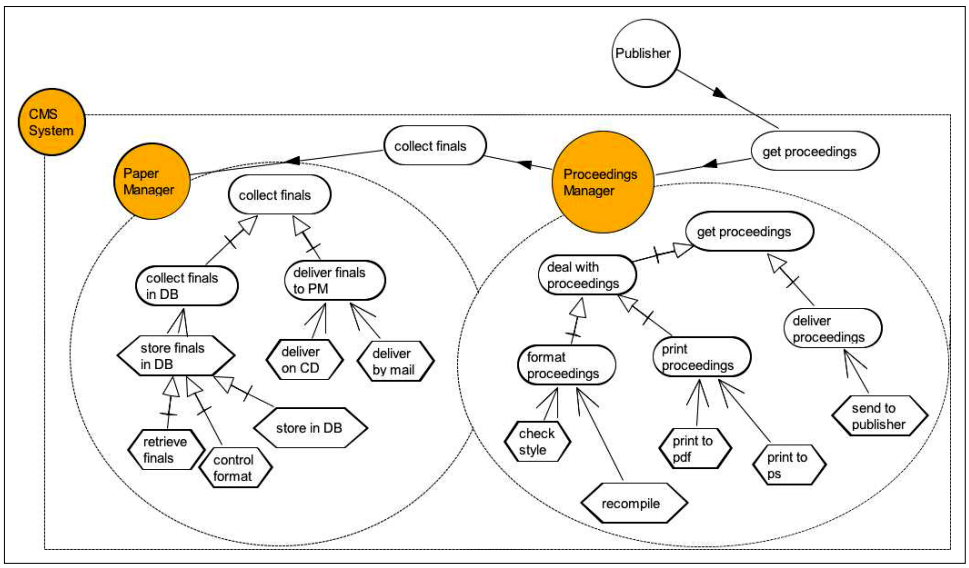
In detailed design, each of the systems allows specification of further details that can be mapped into code fairly straightforwardly, with some automated generation of skeleton code.

5.1 Tropos and TAOM4E

During the *Tropos* Architectural Design phase, the sub-actors and their delegated goals and tasks were modelled. The next phase in *Tropos*, as shown in Table 1, is the Detailed Design phase, which consists of analysing and designing he goal models of these new agents (or sub-actors). As in the Late Requirements phase, the engineer can view the internal design of an agent by opening the balloon that shows the goals delegated to the agent. This allows the engineer to create a new view for each agent in order to analyse the goals delegated to it. An agent's goal model can be further refined by decomposing goals, attaching plans to goals via means-ends relations or defining the set of capabilities possessed by the agent.

Figure 30 shows an excerpt of the goal models for two agents: Paper Manager and Proceedings Manager. As shown, the goal get proceedings has been delegated from the Publisher actor to the Proceedings Manager resulting in a dependency between them. The get proceedings goal is AND-decomposed into two sub-goals within the Proceedings Manager: deal with proceedings, which is further decomposed, and deliver proceedings, which is operationalised by the send to publisher plan. The deal with proceedings goal requires access to all the final papers and thus causes the Proceedings Manager to depend on the Paper Manager to achieve the goal collect finals.

Figure 30 Architectural design: simplified goal model of two sub-actors of CMS (see online version for colours)

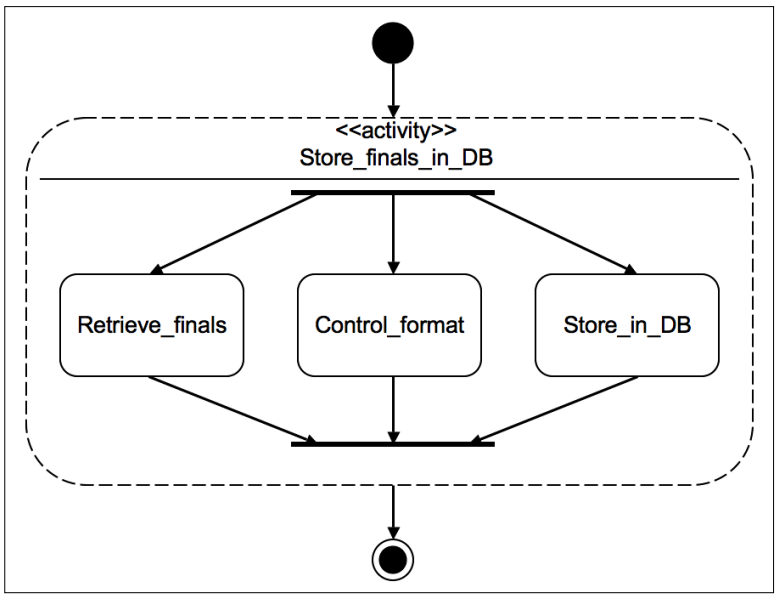


The set of plans possessed by an agent represents its capabilities. As discussed above, *plans* are a means to achieve agent goals that are not delegated to other agents. Defining more than one plan for a goal (as for the goal format proceedings), indicates alternative approaches to achieving a goal. In this instance, one approach to formatting the proceedings is to recompile or reformat them manually, while an alternative approach would be to control the style used by the authors of the posted papers. While the applicability of individual plans is subject to the availability of resources (the source files in this example), the final selection of a plan can also be guided by its positive or negative contribution to soft-goals of interest.

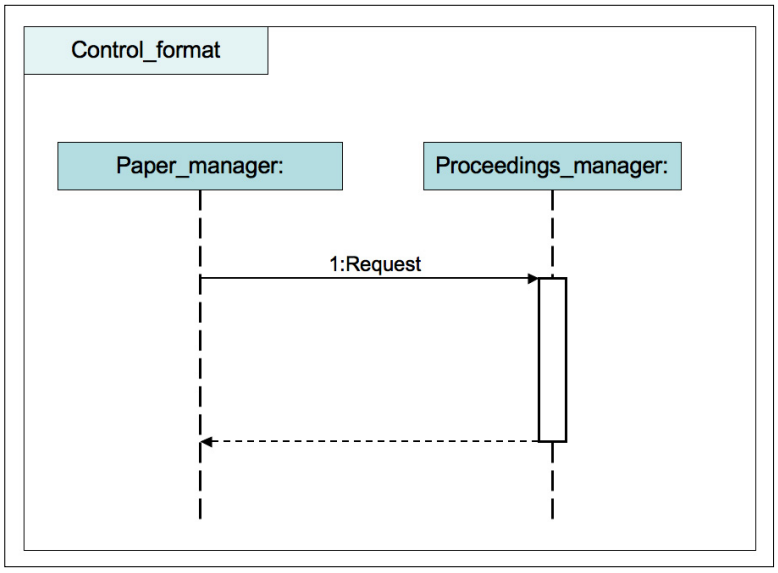
By opening the internal view of the Paper Manager actor, the engineer can see the goal collect finals that was delegated to it from the Proceedings Manager. As shown, collect finals has been decomposed into sub-goals, which are eventually operationalised by plans. These plans can be refined by decomposing them into concrete sub-plans. For example, in Figure 30, the plan store finals in DB has been AND decomposed into retrieve finals, control format and store in DB.

The detailed design is completed by specifying the details of the plans attached to each agent goal and the associated interaction protocols. *Tropos* plans are automatically transformed into UML activity diagrams using the *Tropos2UML* tool. The resulting activity diagram for the plan Store finals in DB is shown in Figure 31(A), which shows that three subplans of store finals in DB (from the Paper Manager in Figure 30) are performed in parallel. Activity diagrams can be further detailed and modified using any UML2 editor capable of reading/writing XMI format. Sequence diagrams, such as Figure 31(B), which shows the protocol used by the activity Control format, are used to define the communications required between agents.

Figure 31 An example of activity (A) and sequence (B) diagrams for the capability internal design (see online version for colours)



(A)



(B)

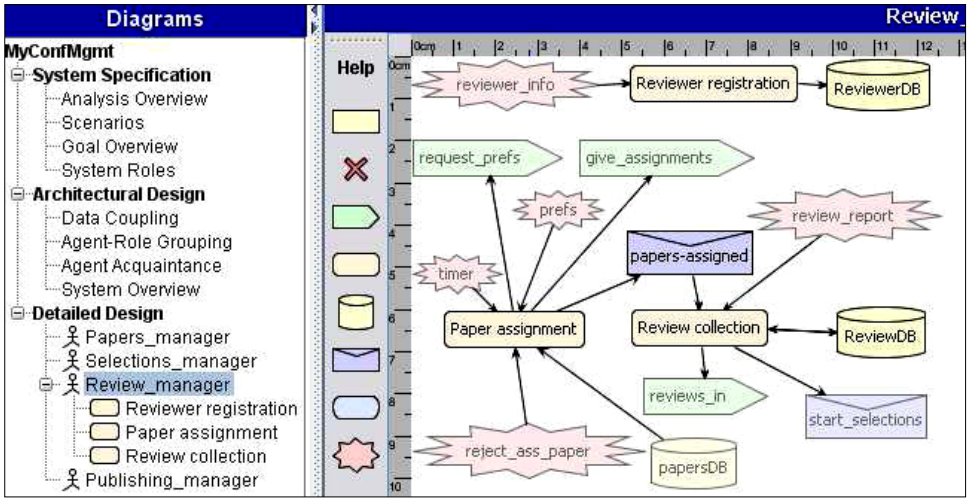
Using these diagrams, *JADE* Behaviour code can be generated to implement the agent's reasoning mechanisms for selecting plans at run-time. The generated code skeletons can be executed on the Jadex platform and exhibit the behaviour specified in the corresponding goal model.

5.2 Prometheus and PDT

The detailed design stage of Prometheus is concerned with design of the agent internals, to allow the agent to achieve the goals associated with it (via its roles and associated goals) and to engage in the interactions specified. A generic stage of detailed design describes agents in terms of capabilities or modules. These capabilities are then finally specified in terms of plans and events, which are of necessity more specific to the implementation paradigm or platform, than the preceding steps. Specification of process diagrams as used in the methodology is not currently supported in PDT.

The detailed design section (bottom left of Figure 32) consists of a list of agent overview diagrams, one for each agent. Each agent has underneath it a list of capability overview diagrams, one for each capability included in the agent. Often the capabilities of the agent will (at least initially)correspond to the roles that were assigned to it, though roles may also be split into multiple smaller capabilities or merged into a larger capability. For example, in this case the Review manager agent had two roles assigned to it (Assignment and review_management) and has three capabilities: 'Reviewer registration', 'Papers assignment' and 'Review Collection'.

Figure 32 Agent overview diagram for reviewer_manager (see online version for colours)

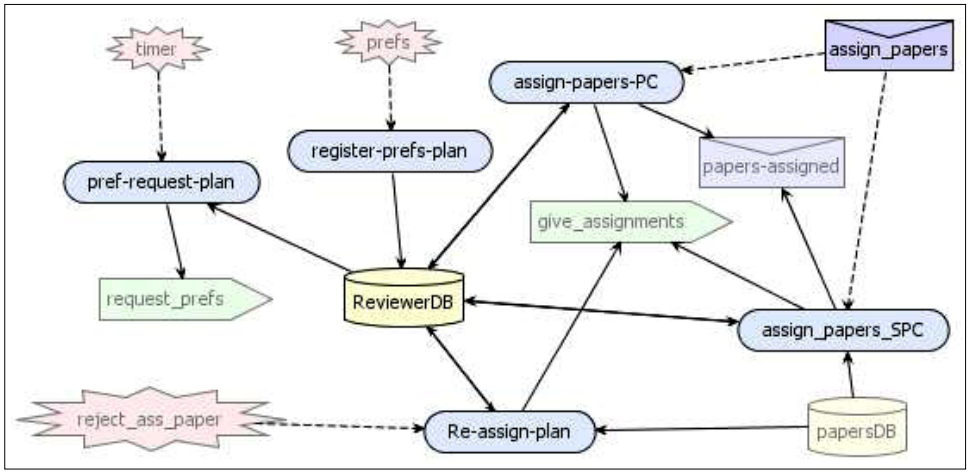


All the entities that were associated with the agent in the system overview diagram are propagated to the agent overview diagram, including the individual messages from protocols associated with the agent. Entities in an agent/capability overview diagram that are propagated form part of the interface to the internals of the agent/capability and are shown as 'faded' icons. These interface entities must then be connected to internal capabilities or plans defined to use or generate them. The designer needs to ensure that all the actions, percepts, messages and data access are accounted for. For example, the 'Reviewer registration' capability handles the percept 'review_info' and modifies data in the 'ReviewerDB'.

Capabilities, which are specified using the 'Capability Overview Diagram', contain the plans that actually do things. Similarly to the agent overview diagram, percepts, messages, actions and data are propagated into this diagram and plans or (sub)capabilities

are created to handle the relevant entities. A dotted line from a percept or message to a plan indicates that the percept/message is the trigger of the plan. Figure 33 outlines the details of the 'Paper assignment' capability. The 'assign-papers-PC' plan is triggered by a message to assign the papers (assign_papers), reads data from 'ReviewerDB' and 'PapersDB', assigns papers to PC members (give_assignments), records the assignments in 'ReviewerDB' and, when all assignments are complete, sends a 'papers-assigned' message.

Figure 33 Capability overview diagram for paper assignment (see online version for colours)



Plan descriptors allow for additional information such as a description of the plan, a context condition specifying the conditions under which this plan is applicable, a failure condition under which the plan may fail, a failure recovery procedure if the plan fails, and a description of the plan body where the developer may specify pseudocode that can be easily translated to code.

As the details of a design are developed, it is very common that one recognises the need for new percepts, actions, messages and so on. This will inevitably lead to the need to revise slightly the models developed at an earlier stage. PDT supports this by automatically introducing any new percepts and actions identified, into the system overview and analysis overview diagrams. Examples of this in the current design are the timer that is the trigger to ask reviewers to indicate which of the submitted papers they would like to review and the reject_ass_paper, which allows a reviewer to reject an assignment with which they have a conflict. These were identified during detailed design and, as a result, were introduced into the System Overview, Analysis Overview and System Roles diagrams. In the System Overview the connections to the appropriate agents were also able to be made. In this case the decision was made not to leave the timer percept in the Analysis Overview or System Roles diagrams, as it did not add to understanding at System Specification level. However, the reject_ass_paper does lead to a fuller understanding of the system functionality and so was connected to the review scenario and the Review_management role. The protocol choose_reviewers should then also be updated to show the role that these two new percepts play in the interaction around assigning reviewers to papers.

Once the detailed design has been completed, it is possible to generate skeleton (JACK) code from the Tools menu. The developer can then add to this code using a text editor. In order to maintain consistency between code and design, any additions or deletions of entities or relationships between entities should be made in the design tool and code regenerated on this basis. Code that is added outside that which is generated by PDT is maintained between design code iterations.

5.3 O-MaSE and aT³

The detailed design of the agents are represented by defining their capabilities, which can be represented as either a set of plans or the definition of the actions within those plans. These plans and action definitions are captured in aT^3 via a set of Plan Models and Capability-Action Models. Neither O-MaSE or aT^3 require any specific agent architecture to be defined. Instead, a code generation engine is being developed for aT^3 that will allow the developer to select the architecture of choice and then produce the code for that architecture based on the diagrams existing in aT^3.

5.3.1 Agent plan model

A plan represents a means by which agents can satisfy a goal in the organisation. Thus a plan can be viewed as an algorithm for achieving a specific goal. Again, because there are four different agents defined in the Agent Class Model, there should be at least four Agent Plan Models developed, one for each agent. Depending on the internal architecture chosen for each agent, the designer could develop multiple Agent Plan Models for each agent. This might be the case when a unique plan is required for each role that an agent could play or if the agent can choose between multiple plans to achieve the same goal. In either case, the agent architecture would be responsible for selecting the appropriate plans and interleaving their execution if required. aT^3 supports the modelling of plans via the Agent Plan Model editor.

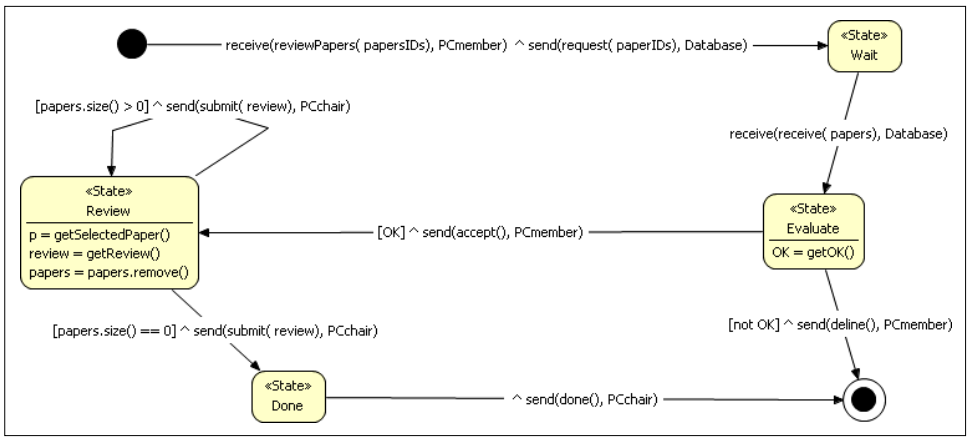
In aT^3, plans are modelled using a finite state automata to specify a single thread of control that defines the behaviour that the agent should exhibit. As such, each plan has a `start state` and an `end state`. All messages are sent and received on state transitions. For the Plan Model, the syntax of the transitions is `[ guard ] receive(message,sender) / send (message,receiver).` The `guard` defines a Boolean condition that determines whether the transition is enabled. The `receive(message,sender)` is a message that is received from the `sender` agent that enables the transition, while the `send(message,receiver)` is a message sent to the `receiver` agent when the transition occurs. Messages are specified in the form *performative* ($p_1...p_n$), where the *performative* is the name of the message and $p_1...p_n$ are the parameters of the message. Each part of the transition is optional and a null transition may exist between two states. aT^3 provides a Transition Properties window for each transition that allows the user to easily edit the components of the transition and ensures correct syntax.

Each state has a (possibly empty) set of actions that are executed sequentially once the state is entered. Each action is represented in the form of a function that returns a value. These actions may represent internal computations of the agent or be part of interactions with objects in the environment. Transitions out of a state are not enabled until all actions have returned their values. The parameters to the actions, the action

return values and all parameters in messages in the plan are considered variables within a single name space. Thus a parameter `X` of a message is the same as the return value of an action `X`. *aT*^{3} provides a State Properties window for each state that allows the user to easily edit the actions for each state.

Figure 34 shows the Plan Model for the Reviewer agent. The plan starts upon receipt of a `reviewpapers` message from the `PCmember` agent. Immediately upon receipt of the message, the agent sends a `request` message to the `Database` agent to get the papers identified by the list of paper identifiers, *paperIDs* and moves into the `Wait` state. When the `Database` returns a list of the *papers* requested, the plan moves into the `Evaluate` state where it interacts with its associated `Reviewer` via the `getOK` action. If the `Reviewer` does not agree to review the set of papers, a `decline` message is sent to the `PCmember` agent and the plan ends. However, if the `Reviewer` does agree to review the set of papers, an `accept` message is sent to the `PCmember` agent and the plan moves to the `Review` state. In the `Review` state, the plan interacts with the `Reviewer` via the `getSelectedPaper` and `getReview` actions. Every time a review is completed, the review is submitted to the `PCchair` agent via a `submit` message and the list of papers is reduced in size. Once the *papers* list is empty, the plan moves into the `Done` state and immediately sends a done message to the `PCchair` agent.

Figure 34 CMS reviewer plan model (see online version for colours)

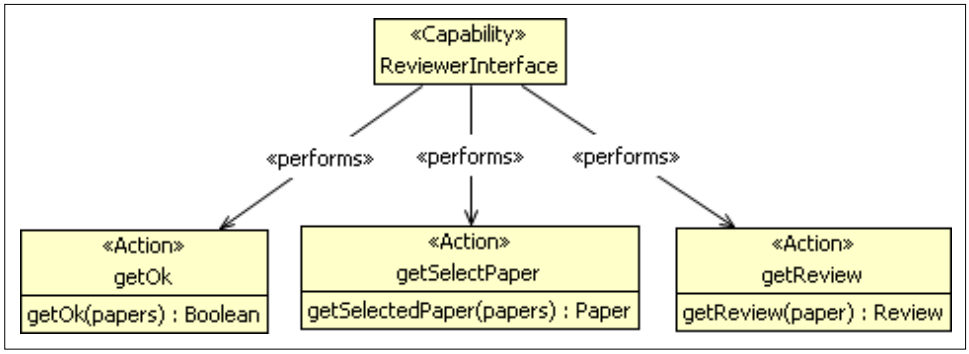


As the Agent Plan Model implements the protocols identified in the Agent Class Model and defined in the Protocol Models, it is critical that a Plan Model be consistent with all Protocol Models that it is required to implement. Thus, by looking at Figure 26, it can seen that `Referee` agent must implement the `reviewPapers`, `getOK`, `writeReview`, `retrievePapers` and `submitReview` protocols. While the `getOK` and `writeReview` protocols interact with the `Reviewer` actor and are implemented as actions, the protocols `reviewPapers`, `retrievePapers` and `submitReview` can be analysed against the `Referee` Agent Plan Model to verify that they are indeed consistent. *aT*^{3} does not currently support automatic verification of the correct implementation of protocols within plans, although plans are in the work to provide that capability using the Bogor model checking framework (Robby *et al.*, 2003).

5.3.2 Capability-action model

The purpose of the Capability-Action Model is to further refine the Capabilities associated with each agent class into a set of actions that the agent plans can use to carry out the details of a plan. Here, in Figure 35, only the `ReviewerInterface` capability and its associated actions are shown. The `ReviewerInterface` capability is *possessed* by the `Referee` agent as shown in Figure 26 and used by the `Reviewer` plan model as shown in Figure 34.

Figure 35 CMS reviewer interface capability-action model (see online version for colours)



Basically, the Capability-Action model defines a set of actions that can be used in a plan to achieve a specific goal. In Figure 35, there are three actions defined: `getOK`, `getSelectedPaper`, `getReview`. Each action is defined in terms of its signature based on objects defined in the Domain Model and a pair of pre- and postconditions (not shown). These pre- and post-conditions define the *intended* effect of the actions on domain objects by accessing and manipulating domain object attributes. aT^3 supports the definition of capabilities and actions via the Capability diagram. aT^3 provides Capability and Action properties panels that allow users to define capability attributes and action signatures.

5.4 Discussion

The goal of Detailed Design is to capture the design of the individual agents identified in the Architectural Design phase. Each of the three methodologies and tools focus on refining each agent based on the goals assigned, either directly or indirectly through roles, to the agents and defining the agent plans required to achieve those goals.

The concepts employed by the three toolkits in the Detailed Design phase are all very similar. Each uses the concepts of goals, plans and capabilities while they use various low-level concepts for defining the internals of the plans such as events, messages, states and actions. Table 8 summarises these concepts for each tool.

All three methods use the notion of a plan as the central concept for representing the low-level behaviour of the agents. *TAOM4E* plan diagrams are used to generate UML activity diagrams. The activity diagrams are supplemented with UML sequence diagrams that define the interactions between agents. This is in contrast to both PDT and aT^3,

which both define the interactions during the Architectural Design phase. PDT plans include a plan description, a pseudocode plan body with conditions under which it may be applied and a failure recovery procedure and conditions when the plan might fail. *aT*^{3} plans are modelled as a finite state machine with actions that are performed in states with message specifications that are sent/received on state transitions.

Table 8 Concepts in the detailed design stage of the three tools

TAOM4E	*PDT*	*aT*^{3}
Agent, plan	Agent, plan	Agent, plan
Goals	Capability, message	Capability, message
	Action, percept, data	Action, state

Each tool also uses the related notion of capabilities, although in slightly different ways. In *TAOM4E*, capabilities are essentially the set of plans the agent can apply. In PDT, capabilities are modules within the agent, the internals of which are then developed to contain plans, events and data. Finally, in *aT*^{3}, agents possess a set of capabilities, which can be refined into plans, or actions, which are typically used to represent actual hardware such as sensors or effectors.

Table 9 summarises the principal models used to describe the internal structure of the agents and their capabilities.

Table 9 Models in the detailed design phase of the three tools

TAOM4E	*PDT*	*aT*^{3}
Agent goal	Agent overview	Agent plan model
Capability's activity and sequence diagram	Capability overview	Capability model

TAOM4E uses an agent-specific goal model to decompose the system level goal and to specify which plans map to those goals. PDT uses an Agent Overview Diagram to connect the previously defined agent interface to internal agent capabilities.

The capabilities are defined using a varied set of models. *TAOM4E* uses UML activity and sequence diagrams to specify agent plans and their interactions. PDT uses capability overview diagrams to show the relationship between plans and internal and external resources, messages and events; the details of the plans are specified textually. *aT*^{3} capabilities can be either plans or actions. Plans are specified in a detailed finite state machine where actions are performed within the states while messages can be sent or received on transitions. Actions, which can represent pre-defined software operations or hardware specific sensors/effectors, are further refined in Capability-Actions diagrams.

Finally, each of the toolsets supports the generation of agent code skeletons on various agent platforms.

6 Other tool support aspects

To round out our discussion of the three different tools presented, we now briefly explore additional functionalities which are not necessarily directly related to a particular design phase presented here. In some cases these functionalities are integrated within the tools, while in others they are provided by associated tools that operate on the output of the tools presented.

6.1 Tropos and TAOM4E

During the *Tropos* development process, other tools can be used in conjunction with *TAOM4E*. These tools include code generators that automatically produce agent code and automated testing tools.

6.1.1 Code generation

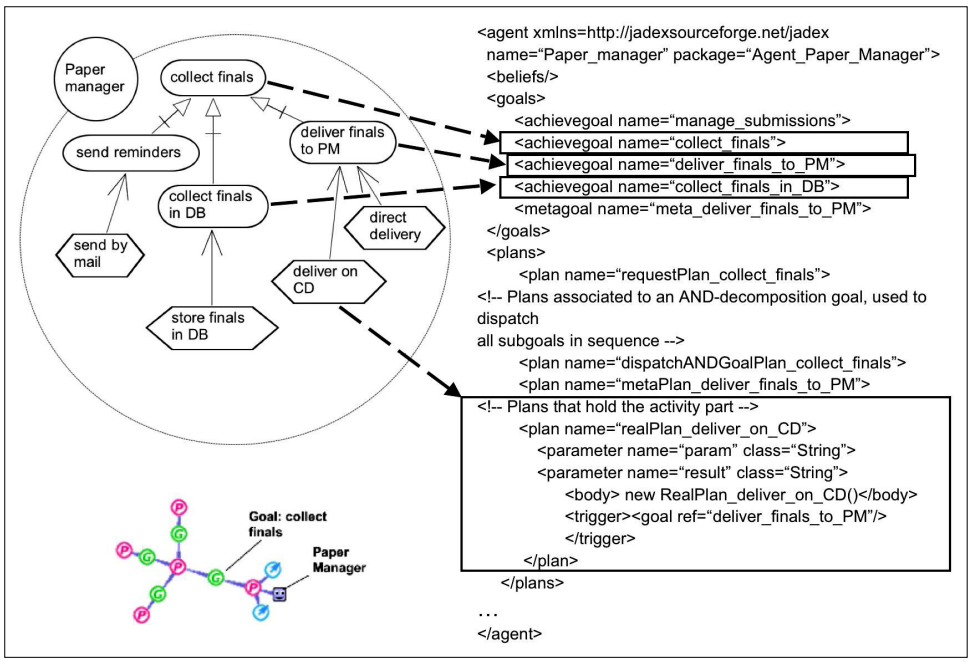
The goal models created in the design phase are the basis for implementing the software agents. Of prime interest is the *knowledge level*, which is the part of an agent responsible for selecting the appropriate plans in order to achieve the desired goals. In an agent's goal model, the *knowledge level* consists of the agent's goals and their decomposition, contributions, dependencies to other agents and means-end relations to plans. This knowledge is input to the *t2x* (*Tropos* to *Jadex*) component, which generates BDI agent skeletons that are executable on the Jade BDI agent platform (Pokahr *et al.*, 2005). The mapping between *Tropos* goal model elements and *Jadex* constructs is described in (Penserini *et al.*, 2007a; Morandini, 2006).

The generated BDI code skeleton implements the reasoning part of a software agent, which consists of an *Agent Definition File* (ADF), in XML format. The ADF defines the goals, plans, beliefs and messages for every system agent defined in the goal model. Plans are implemented in Java and are linked to agents via the ADF. The *t2x* tool is used to generate *Jadex* ADFs by simply selecting a system agent in the goal model and starting the automatic generation process. For the CMS system, code was generated for the two system agents, the Proceedings Manager agent and Paper Manager agent.

The *t2x* tool analyses an agent's goal model by exploring its goal decomposition hierarchy. The goal hierarchy is converted to *Jadex* goals and the Java code that implements the decomposition logic in the form of decomposition graphs. Plans are implemented as Java code and are connected to their related goals by a *triggering* mechanism. These goal decomposition graphs, together with the contributions to soft-goals and dependencies with other agents, are stored in the agent's belief base, which allows the agent to control its run-time behaviour by navigating the goal graph. The generated code skeleton is executable on the *Jadex* platform and can be modified and customised as needed. In particular, the agent code can be extended with code generated by *UML2JADE* tool, which generates code for the activity diagrams specified during the detailed design.

Figure 36 shows *Jadex* XML code for the Paper Manager agent, which corresponds to the goal model on the top-left side of the figure. A visualisation generated by the *Introspector* tool on the *Jadex* platform for the goal model's run-time reasoning trace is shown on the bottom-left.

Figure 36 Simplified goal diagram for Paper Manager, part of generated *Jadex* XML code (the code corresponding to three goals and a plan is highlighted), and example *Jadex* run-time agent instance with activated goals and plans, visualised by the *Introspector* tool provided by the *Jadex* platform (see online version for colours)

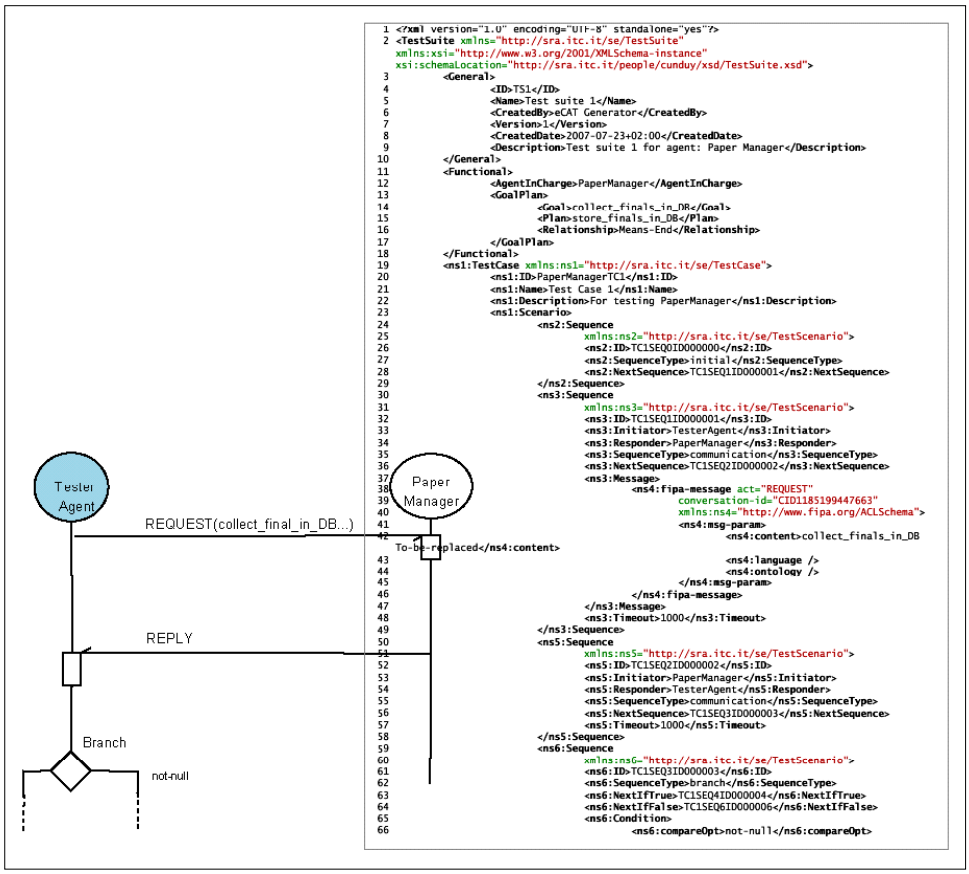


6.1.2 Testing

Tropos analysis and design is complemented by testing activities that follow the goal-oriented testing methodology presented in (Nguyen *et al.*, 2007). The *eCAT*[13] Eclipse plugin helps derive test suites from goal diagrams and can be integrated with *TAOM4E*. The *eCAT* tool allows testers to define test inputs and oracles as well as to automatically generate and evolve additional test inputs during the course of testing. *eCAT* runs these test inputs continuously to extensively stress the system under test (Nguyen *et al.*, 2008a–b).

The eCAT tool generates test suites for every elementary relationship between a goal and a plan. The test suite is used to guide the *Autonomous Tester Agent*, which triggers the appropriate goals in order to verify the execution of the corresponding plan. In the case of CMS, *eCAT* takes the architectural diagram from Figure 30 as an input and generates a set of test suites for each agent. When generating test suites, developers can choose which communication protocols the *Autonomous Tester Agent* will use to communicate with the agents. Figure 37 illustrates a test suite for testing whether the Paper Manager agent is able to achieve the goal collect finals in DB. The graphical part of the figure gives an intuitive understanding of the test suite, which is formalised in XML. When executing tests, the *Autonomous Tester Agent* sends a 'REQUEST' message along with the goal name collect finals in DB to the Paper Manager. It then waits for a reply and decides whether to finish the test or to continue with other requests.

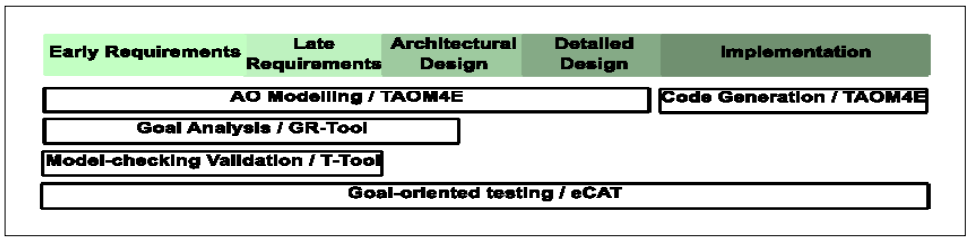
Figure 37 Example of a test scenario. An excerpt of the XML specification is depicted in the right part (see online version for colours)




```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TestSuite xmlns="http://sra.itc.it/se/TestSuite"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://sra.itc.it/people/cunduy/xsd/TestSuite.xsd">
        <General>
                <ID>TS1</ID>
                <Name>Test suite 1</Name>
                <CreatedBy>eCAT Generator</CreatedBy>
                <Version>1</Version>
                <CreatedDate>2007-07-23+02:00</CreatedDate>
                <Description>Test suite 1 for agent: Paper Manager</Description>
        </General>
        <Functional>
                <AgentInCharge>PaperManager</AgentInCharge>
                <GoalPlan>
                        <Goal>collect_finals_in_DB</Goal>
                        <Plan>store_finals_in_DB</Plan>
                        <Relationship>Means-End</Relationship>
                </GoalPlan>
        </Functional>
        <ns1:TestCase xmlns:ns1="http://sra.itc.it/se/TestCase">
                <ns1:ID>PaperManagerTC1</ns1:ID>
                <ns1:Name>Test Case 1</ns1:Name>
                <ns1:Description>For testing PaperManager</ns1:Description>
                <ns1:Scenario>
                        <ns2:Sequence
                                xmlns:ns2="http://sra.itc.it/se/TestScenario">
                                <ns2:ID>TC1SEQ0ID000000</ns2:ID>
                                <ns2:SequenceType>initial</ns2:SequenceType>
                                <ns2:NextSequence>TC1SEQ1ID000001</ns2:NextSequence>
                        </ns2:Sequence>
                        <ns3:Sequence
                                xmlns:ns3="http://sra.itc.it/se/TestScenario">
                                <ns3:ID>TC1SEQ1ID000001</ns3:ID>
                                <ns3:Initiator>TesterAgent</ns3:Initiator>
                                <ns3:Responder>PaperManager</ns3:Responder>
                                <ns3:SequenceType>communication</ns3:SequenceType>
                                <ns3:NextSequence>TC1SEQ2ID000002</ns3:NextSequence>
                                <ns3:Message>
                                        <ns4:fipa-message act="REQUEST"
                                                conversation-id="CID1185199447663"
                                                xmlns:ns4="http://www.fipa.org/ACLSchema">
                                                <ns4:msg-param>
                                                        <ns4:content>collect_finals_in_DB
To-be-replaced</ns4:content>
                                                        <ns4:language />
                                                        <ns4:ontology />
                                                </ns4:msg-param>
                                        </ns4:fipa-message>
                                </ns3:Message>
                                <ns3:Timeout>1000</ns3:Timeout>
                        </ns3:Sequence>
                        <ns5:Sequence
                                xmlns:ns5="http://sra.itc.it/se/TestScenario">
                                <ns5:ID>TC1SEQ2ID000002</ns5:ID>
                                <ns5:Initiator>PaperManager</ns5:Initiator>
                                <ns5:Responder>TesterAgent</ns5:Responder>
                                <ns5:SequenceType>communication</ns5:SequenceType>
                                <ns5:NextSequence>TC1SEQ3ID000003</ns5:NextSequence>
                                <ns5:Timeout>1000</ns5:Timeout>
                        </ns5:Sequence>
                        <ns6:Sequence
                                xmlns:ns6="http://sra.itc.it/se/TestScenario">
                                <ns6:ID>TC1SEQ3ID000003</ns6:ID>
                                <ns6:SequenceType>branch</ns6:SequenceType>
                                <ns6:NextIfTrue>TC1SEQ4ID000004</ns6:NextIfTrue>
                                <ns6:NextIfFalse>TC1SEQ6ID000006</ns6:NextIfFalse>
                                <ns6:Condition>
                                        <ns6:compareOpt>not-null</ns6:compareOpt>
```

Other tool-supported analysis techniques are available in *Tropos*. These techniques include formal analysis on requirements and system design goal models via the GR-Tool (Giorgini *et al.*, 2005) as well as the previously mentioned validation of requirements specification via model-checking with the T-Tool (Fuxman *et al.*, 2001). These types of analysis are particularly useful with complex models. Figure 38 shows the tools that support the various activities and phases of the Tropos process. The Agent-Oriented modelling activity spans the first four phases and is completely supported by *TAOM4E* while the implementation phase is supported by the *TAOM4E* generator components. The GR-tool (Giorgini *et al.*, 2005) and T-Tool support reasoning and validation activities during the early phases of the development while the *eCAT* component supports continuous testing over the entire process.

Figure 38 *Tropos* development process phases: activities and supporting tools (see online version for colours)



6.2 Prometheus and PDT

PDT has a number of additional tools, or extended versions, that offer capabilities not yet integrated into the mainstream version of PDT. In addition, it has a number of features which are important for a software engineering support tool but which are not necessarily agent-specific, Prometheus-specific or related to a particular phase of the design process. For example, the user interface will continuously prevent the following sorts of errors:

1 Definition: it is not possible to have references to non-existent entities, since creating a reference will create the entity if it does not exist and when an entity is deleted all references to it are deleted as well.

2 Naming: it is not possible for two entities to have the same name, for example a goal and a plan both called ‘assign-Papers-PC’.

3 Simple type errors: for example, it is not possible in PDT to connect an action and another action.

4 Scope constraints: for example, it is not possible to create an incoming percept to a plan without that percept also being:

   - shown on the system overview diagram
   - shown as incoming to the agent whose plan it is.

5 Violations of interface declarations: for example, if an agent is specified as reading a belief set, then it is not possible to create an arrow from one of the agent’s plans to the belief set. Similarly, if an agent specifies that it only sends a message, then its plans cannot receive the message. PDT does not allow the user to violate this constraint.

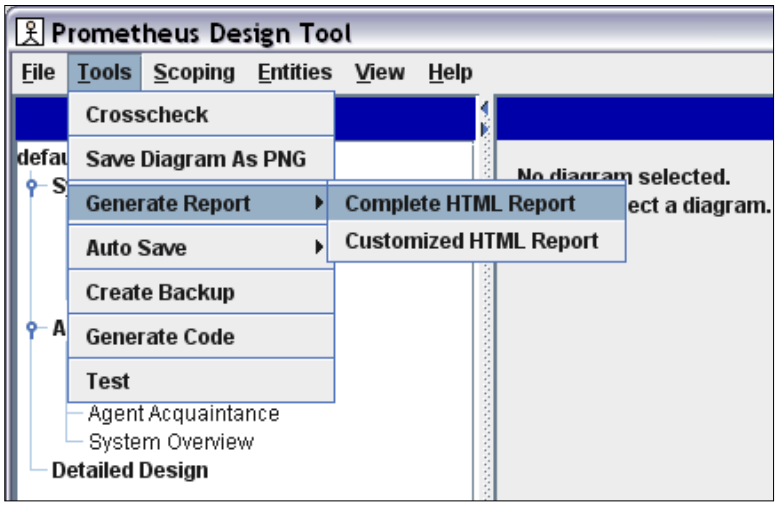
PDT also has a number of additional features available from the tools menu shown in Figure 39. These build on the particular characteristics of agent designs and, while they are specific to PDT and Prometheus, many of them could readily be adapted in principle to other methodologies and tools. Some of the additional features that PDT provides include:

- Crosschecking – this is a consistency check that is performed on demand, generating a list of errors and warnings that can be checked by the developer. Examples of a warning are writing of internal data that are never read, while an example of an error is a mismatch between the interaction protocol specified between two agents and the messages actually sent and received by processes within those agents.

- Code generation – the detailed design specification is close to code and the tool currently provides a code generation feature that generates skeleton code of the system in the JACK agent language (Busetta *et al.*, 1998). The skeleton code can then be completed by the developers. The tool supports repeated code generation from the design, preserving any user-edited code segments.
- Report generation – one of the very useful features of the tool is its ability to generate an HTML design document. This document contains both figures and textual information, as well as an index over all the design entities. The report can also be customised such that only certain entities are included in the report. The tool can also save printable images of the various diagrams (in PNG format).
- Auto save and backup – PDT can be set to automatically save the current project at a set time interval (which can be changed) and also allows for creating backup files, which save the current version into a different file specified by the user.

PDT is also available with an Eclipse plugin, enabling it to be used within a broader IDE, supporting aspects such as syntax highlighting, version management and so on. Details are available from the PDT home page.[14]

Figure 39 Tools in PDT (see online version for colours)



There are also a number of separate tools that can take PDT produced files as input or that are extended versions of PDT. Some of these are in the process of incorporation into the main version of the software. They include:

- Automated unit testing: we have a prototype tool that does fully automated generation and execution of test cases for plans, events and beliefs within an agent, based on the design model (Zhang *et al.*, 2007). The tool provides a report to the user. Users can interact with the tool to specify additional test cases, and test case libraries can be maintained. This is currently being integrated with the public version of PDT and will be released shortly.

- Design-driven debugger: we have a prototype debugging tool and framework that uses the design documents produced by PDT to identify and provide alerts regarding errors. This has been evaluated in a thorough user study and shown to provide substantial help in detecting errors (Poutakidis *et al.*, 2002; Padgham *et al.*, 2005).
- Model-based code generation: we have an extended version of PDT that supports more detailed but also more constrained, models that are sufficient for automated generation of fully operational code. This was developed primarily to allow modification of a system by domain experts in an application area. It has been evaluated in the meteorology domain (Jayatilleke *et al.*, 2005a–b).
- Priority-based incremental development: we have developed mechanisms for taking high level priorities on scenarios and propagating these in a coherent manner through the system, to support guided incremental development of functionality (Padgham and Perepletchikov, 2007). This was incorporated in an earlier version of PDT but needs updating into the current version.
- Maintenance support at the design level: we have prototype tools which assist a developer when making changes required for new releases during the life of a system (the maintenance phase). These use formally specified Object Constraint Language (OCL) constraints and a metamodel to assist the engineer to make suitable secondary changes to return the design to a consistent state following a primary change. This work is presented in (Dam and Winikoff, 2008).

6.3 O-MaSE and aT³

In addition to the model plugins, *aT*³ also provides several features to better supports developers. Specifically, *aT*³ includes the following features:

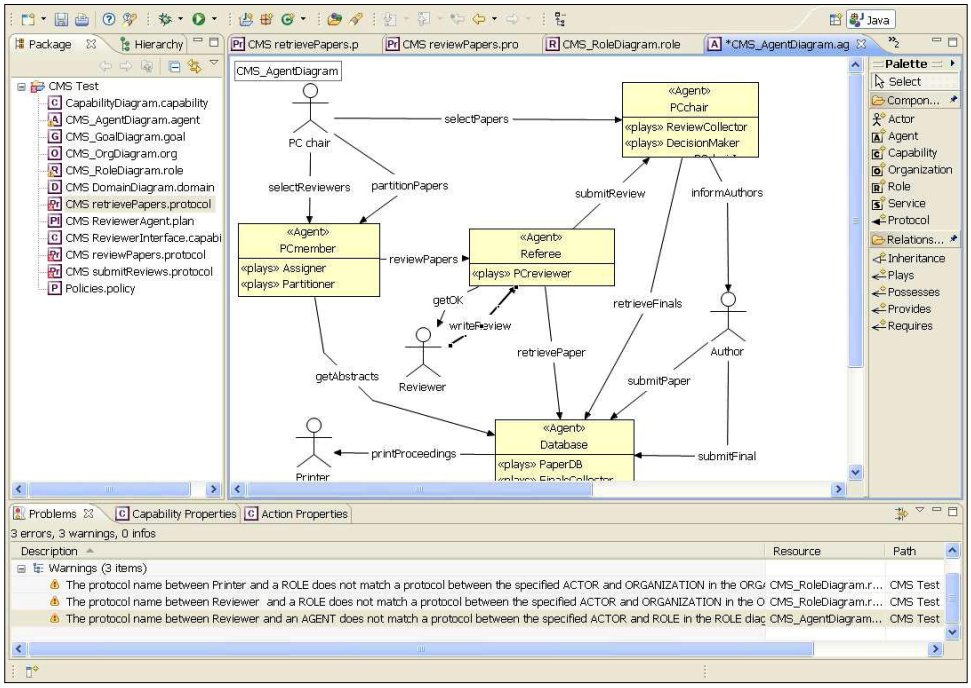
- the agentTool Process Editor – a process definition and verification tool
- the *aT*³ Verification Framework – a tool for verifying consistency in and between models
- a suite of predictive metrics computed using model checking techniques
- a code generation capability.

A unique feature to *aT*³ is the inclusion of the *agentTool Process Editor* (APE) plugin, which was developed to support the design and definition of O-MaSE compatible processes. As discussed in Garcia-Ojeda *et al.* (2007), O-MaSE is actually a framework that helps process engineers define custom multi-agent systems development processes. O-MaSE implements a Method Engineering approach to process construction. The APE plugin is based on the Eclipse Process Framework and provides an Eclipse perspective for designing O-MaSE compliant processes. A process designer may use the APE plugin to extend O-MaSE with new tasks, models or usage guidelines. The plugin also provides the ability to create new process instances by selecting various tasks, models and producers from the O-MaSE method fragment library and then verify that they meet process guidelines.

Once a process designer has developed a process for a specific use, the designer may verify that it is O-MaSE compliant, via the aT^3 APE verification tool. Basically, the APE verification tool checks to ensure that the task ordering as defined by the process designer is valid and that each task has the appropriate inputs. Warnings and error messages are displayed to the designer using the familiar Eclipse Problems view.

The aT^3 *Verification Framework* plugin uses a set of rules to check model validity as well as consistency between models. This verification checking is done in real-time with warnings and errors displayed in the Eclipse Problem view in a fashion similar to Eclipse compiler warnings and errors as shown in Figure 40. The Verification Framework allows for the easy addition of new rules as well as the ability for the user to turn rules on and off as required.

Figure 40 AgentTool verification framework (see online version for colours)



The verification plugin runs in the background every time a diagram is saved. The verification plugin has a number of rules that it runs against the current model, which may require loading in data from other saved model diagrams as well. Thus, the verification plugin keeps an updated version of all related models in memory in order to verify the current model. Figure 40 shows the case where a leaf goal, `informAccepted`, has not been assigned to a role in the role model. Here `informAccepted` should be assigned to be played by the `DecisionMaker` role. Some other examples of the errors and warnings handled for the various models include:

- Each leaf goal in a Goal Model should be 'achieved' by some role in the Role Model.
- Each agent connected to a role by a 'plays' relation must also be connected to every capability that the role is connected to by a 'requires' relation in the role diagram by a 'possess' relation.
- Each role in an Agent Class Model should exist in a Role Model in the current directory.
- No two agents may be connected via a 'protocol' relation unless the roles they connect to with the 'plays' relation are themselves connected by a 'protocol' relation in a Role Model.

There are also a current effort to develop plugin support for a set of *predictive metrics* using the Bogor model checker (Robby *et al.*, 2003). The idea of these predictive metrics is to provide design-time support that predicts how a multi-agent system will behave at run-time. The metrics for measuring system flexibility and the importance of specific goals to the overall system have already been defined. The envisioned plugins will allow the designer to literally push a button and get feedback predicting how the system will operate within a set of user-defined assumptions.

Finally, a *Code Generation Framework* plugin is being developed that will allow aT^3 to generate code for a variety of architecture/platform combinations. The first plugin has been focused towards the Cooperative Robotic Organisation System (CROS) simulator, which was developed at Kansas State University. While the current code generated is specific to the CROS simulator and the OMACS agent architecture, a pattern-based approach is being used that will make the framework easily extensible to other platforms and architectures.

6.4 Discussion

Again there is quite a lot of similarity between the tools described here but also some differences. Both *Tropos* and Prometheus support code generation while this is also underway in O-MaSE. Prometheus also has an extension that supports fully automated model driven code generation, using more detailed model specifications. *Tropos* provides support tools for formal validation and also for formal analysis of the goal model. Prometheus and O-MaSE on the other hand provide built in rule-based consistency checking and verification within and between models. Prometheus has the ability to generate a design document and also has a separate prototype debugging tool that uses the design models. Both Prometheus and *Tropos* have automated testing tools based on the design models. O-MaSE is unique in having a process editor that allows composition of models. O-MaSE is also in the process of providing a plug-in for predictive metrics, while Prometheus has a version of PDT that provides support for incremental development by propagating priorities.

Table 10 compares the additional areas of work and functionality of the different approaches.

Table 10 Additional functionalities, prototypes and plug-ins for the three methodologies

Tropos	*Prometheus*	*O-MaSE*
Automated testing	Automated testing	
Separate formal validation tool	Rule based consistency checks	Rule based consistency checks
Code generation	Code generation	Code generation in process
	Design document generation	
	Prototype debugging tool	
	Priority propagation prototype	
		Process editor
		Predictive metrics (in process)

7 Conclusion and future work

This paper has explored in detail the design process of a Conference Management System, using the toolkits of three different methodologies. All three methodologies (and toolkits) are ongoing active research projects, exploring the kinds of advanced support that is needed and can be provided for the design and development of agent systems. This paper, and the workshop session from which it arose, has been a concerted effort to understand and compare the similarities and differences between the toolkits and the underlying approaches. It is clear that although there are differences, there is substantial similarity, and the core underlying design issues and questions that are being supported are substantially similar. Some work has already been done between these authors and others on discussing potential notation standardisation, which would be a helpful first step for users to more readily be able to compare design artefacts across the methodologies.

As the field of support tools for Agent Oriented Software Engineering matures within the academic domain, it is to be hoped that the core ideas may be taken up by companies willing to provide commercial tools. There is still much useful work to be done in such areas as metrics, maintenance support tools, more advanced testing and debugging, formal verification and other such areas.

Acknowledgements

The work associated with Prometheus was supported by the Australian Research Council (ARC) and Agent-Oriented Software, under grant LP0453486 ‘Advanced Software Engineering Support for Intelligent Agent Systems’. The work associated with O-MaSE was supported by grants from the US National Science Foundation (0347545) and the US Air Force Office of Scientific Research (FA9550-06-1-0058). The work associated with *Tropos* was supported by the STAMPS project financed by the Autonomous Province of Trento, Italy.

References

Bergenti, F., Gleizes, M-P. and Zambonelli, F. (Eds.) (2004) *Methodologies and Software Engineering for Agent Systems. The Agent-Oriented Software Engineering Handbook*, Kluwer Publishing, ISBN: 1-4020-8057-3.

Bertolini, D., Novikau, A., Susi, A. and Perini, A. (2006) 'TAOM4E: an eclipse ready tool for agent-oriented modeling. Issue on the development process', Technical report, Fondazione Bruno Kessler – irst.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. and Mylopoulos, J. (2004) 'Tropos: an agent-oriented software development methodology', *Autonomous Agents and Multi Agent Systems*, Vol. 8, No. 3, pp.203–236.

Busetta, P., Rönnquist, R., Hodgson, A. and Lucas, A. (1998) 'JACK intelligent agents – components for intelligent agents in Java', Technical report, Melbourne, Australia: Agent Oriented Software Pty. Ltd., http://www.agent-software.com.

Ciancarini, P., Niestrasz, O. and Tolksdorf, R. (1998) 'A case study in coordination: conference management on the internet', ftp://cs.unibo.it/pub/cianca/coordina.ps.gz., citeseer.ist.psu.edu/ciancarini98case.html.

Ciancarini, P., Omicini, A. and Zambonelli, F. (1999) 'Multiagent system engineering: the coordination viewpoint', in N. Jennings and Y. Lesperance (Eds.) *6th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Berlin: Springer-Verlag, Vol. 1757, pp.250–259, citeseer.ist.psu.edu/article/ciancarini00multiagent.html.

Dam, K. and Winikoff, M. (2003) 'Comparing agent-oriented methodologies', *Proceedings of the 5th Int'l Bi-Conference Workshop on AgentOriented Information Systems (AOIS)*, Melbourne, Australia.

Dam, K.H. and Winikoff, M. (2008) 'Cost-based bdi plan selection for change propagation', *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'08)*, pp.217–224.

DeLoach, S.A. (2001) 'Analysis and design using MaSE and agentTool', *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, Miami University, Oxford, Ohio, 31 March–1 April.

DeLoach, S.A. (2002) 'Modeling organizational rules in the multi-agent systems engineering methodology', *AI '02: Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, London, UK: Springer-Verlag, pp.1–15.

Fuxman, A., Liu, L., Mylopoulos, J., Roveri, M. and Traverso, P. (2004) 'Specifying and analyzing early requirements in tropos', *Requir. Eng.*, Vol. 9, No. 2, pp.132–150.

Fuxman, A., Pistore, M., Mylopoulos, J. and Traverso, P. (2001) 'Model checking early requirements specifications in Tropos', *IEEE Int. Symposium on Requirements Engineering*, Toronto, Canada: IEEE Computer Society, pp.174–181.

Garcia-Ojeda, J., DeLoach, S.A., Robby and Valenzuela, J. (2007) 'O-MaSE: a customizable approach to developing multiagent development processes', *Agent Oriented Software Engineering VIII (AOSE'07)* (postproceedings, 2007), LNCS, Springer.

Giorgini, P., Mylopoulos, J., Perini, A. and Susi, A. (2008) 'The Tropos methodology and software development environment', in P. Giorgini, N. Maiden, J. Mylopoulos and E. Yu (Eds.) *Social Modelling for Requirements Engineering*, MIT Press.

Giorgini, P., Mylopoulos, J. and Sebastiani, R. (2005) 'Goal-oriented requirements analysis and reasoning in the tropos methodology', *Engineering Applications of Artificial Intelligence*, Vol. 18, No. 2, pp.159–171.

Henderson-Sellers, B. and Giorgini, P. (Eds.) (2005) *Agent-Oriented Methodologies*, Idea Group Publishing.

Huget, M-P. and Odell, J. (2004) 'Representing agent interaction protocols with agent UML', *Proceedings of the Fifth International Workshop on Agent Oriented Software Engineering (AOSE)*, http://www.jamesodell.com/aose2004.

Jayatilleke, G. (2007) 'A model driven component agent framework for domain experts', PhD thesis, RMIT University, School of Computer Science and Information Technology.

Jayatilleke, G.B., Padgham, L. and Winikoff, M. (2005a) 'A model driven component-based development framework for agents', *Computer Systems Science and Engineering*, Vol. 4, No. 20.

Jayatilleke, G.B., Padgham, L. and Winikoff, M. (2005b) 'Component Agent Framework for Non-experts (CAFnE) toolkit', *Software Agent-Based Applications, Platforms and Development Kits*, Birkhäuser, ISBN: 3-7643-7347-4, pp.169–195.

Kolp, M., Giorgini, P. and Mylopoulos, J. (2003) 'Organizational patterns for early requirements analysis', *Proceedings of CAiSE*, Springer, Vol. 2681 of Lecture Notes in Computer Science, pp.617–632.

Luck, M. and Padgham, L. (Eds.) (2008) *Agent Oriented Software Engineering VIII (AOSE'07)*, Springer, Vol. 4951 of LNCS.

Miller, M. (2007) 'A goal model for dynamic systems', Technical report, Kansas State University, Computer Science Department, Masters Thesis.

Morandini, M. (2006) 'Knowledge level engineering of BDI agents', Master's thesis, Department of Computer Science, University of Trento, Italy, http://dit.unitn.it/~morandini/resources/ThesisMirkoMorandini.pdf.

Nguyen, C.D., Perini, A. and Tonella, P. (2008a) 'Constraint-based evolutionary testing of autonomous distributed systems', *Proc. of the International Workshop on Search-Based Software Testing (SBST)*, 9–11 April.

Nguyen, C.D., Perini, A. and Tonella, P. (2008b) 'Ontology-based test generation for multi agent systems', *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, 12–16 May, pp.1315–1318.

Nguyen, D.C., Perini, A. and Tonella, P. (2007) 'A goal-oriented software testing methodology', *8th International Workshop on Agent-Oriented Software Engineering, AAMAS*, http://sra.itc.it/people/cunduy/publications/gotesting-cmr.pdf.

Padgham, L. and Perepletchikov, M. (2007) 'Prioritisation mechanisms to support incremental development of agent systems', *IJAOSE*, Vol. 1, Nos. 3–4, pp.477–497.

Padgham, L. and Winikoff, M. (2004) *Developing Intelligent Agent Systems: A Practical Guide*, John Wiley and Sons, ISBN: 0-470-86120-7.

Padgham, L., Winikoff, M. and Poutakidis, D. (2005) 'Adding debugging support to the prometheus methodology', *Journal of Engineering Applications in Artificial Intelligence*, Vol. 18, No. 2.

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J. (2006) 'From stakeholder intentions to software agent implementations', *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE'06)*, Luxemburg: Springer-Verlag, Vol. 4001 of LNCS, pp.465–479.

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J. (2007a) 'From stakeholder intentions to agent capabilities', *Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07)*, Haway, USA: ACM Press.

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J. (2007b) 'High variability design for software agents: extending Tropos', *ACM TAAS*, Vol. 2, No. 4.

Perini, A. and Susi, A. (2005) 'Agent-oriented visual modeling and model validation for engineering distributed systems', *Computer Systems Science & Engineering*, Vol. 20, No. 4, pp.319–329.

Pokahr, A., Braubach, L. and Lamersdorf, W. (2005) 'Jadex: a bdi reasoning engine', in J.D.R. Bordini, M. Dastani and A.E.F. Seghrouchni (Eds.) *Multi-Agent Programming*, USA: Springer Science+Business Media Inc., pp.149–174, book chapter.

Poutakidis, D., Padgham, L. and Winikoff, M. (2002) 'Debugging multi-agent systems using design artifacts: the case of interaction protocols', *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02)*, Bologna, Italy, pp.960–967.

Robby, Dwyer, M.B. and Hatcliff, J. (2003) 'Bogor: an extensible and highly-modular software model checking framework', *'ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*', New York, NY: ACM, pp.267–276.

Santos, D., Blois, M. and Bastos, R. (2007) 'Developing a conference management system with the multi-agent systems unified process a case study', *8th International Workshop on Agent-Oriented Software Engineering*, Honolulu, Hawaii, USA, pp.212–224.

Sierra, C., Thangarajah, J., Padgham, L. and Winikoff, M. (2007) 'Designing institutional multi-agent systems', in L. Padgham and F. Zambonelli (Eds.) *Agent Oriented Software Engineering VII: 7th International Workshop, AOSE 2006*, LNCS, Springer-Verlag, pp.84–103.

Sudeikat, J., Braubach, L., Pokahr, A. and Lamersdorf, W. (2004) 'Evaluation of agent-oriented software methodologies: examination of the gap between modeling and platform', in P. Giorgini, J. Müller and J. Odell (Eds.) *Agent Oriented Software Engineering*, New York, USA, July.

Van Lamsweerde, A., Letier, E. and Darimont, R. (1998) 'Managing conflicts in goal-driven requirements engineering', *IEEE Trans. Softw. Eng*, Vol. 24, No. 11, pp.908–926.

Winikoff, M. (2007) 'Defining syntax and providing tool support for agent UML using a textual notation', *Int. J. Agent-Oriented Software Engineering*, Vol. 1, No. 2, pp.123–144.

Yu, E. (1995) 'Modelling strategic relationships for process reengineering', PhD thesis, University of Toronto, Department of Computer Science, University of Toronto.

Zambonelli, F., Jennings, N.R. and Wooldridge, M. (2001) 'Organizational abstractions for the analysis and design of multi-agent system', *First International Workshop, AOSE 2000 on Agent-Oriented Software Engineering*, Secaucus, NJ: Springer-Verlag New York, Inc., pp.235–251.

Zhang, Z., Thangarajah, J. and Padgham, L. (2007) 'Automated unit testing for agent systems', *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-07)*, pp.10–18.

Notes

1 An additional system Multi-agent Systems Unified Process (MASUP) was also presented but is not included in this paper (Santos *et al.*, 2007).

2 http://se.fbk.eu/en/tools

3 http://www.eclipse.org

4 http://www.eclipse.org/emf

5 http://www.eclipse.org/gef

6 http://tefkat.sourceforge.net

7 http://www.cs.rmit.edu.au/agents/pdt/

8 At the time of writing the PDT plugin runs under Eclipse version 3.2 and higher. There is a commitment to maintain compatibility with new versions of Eclipse where possible.

9 http://agenttool.cis.ksu.edu/

10 Although it should be noted that in other approaches, discussion of system-related characteristics would be called design.

11 These may be humans or other software systems.

12 Reviewers are actors external to the system and are not shown on the system overview diagram.

13 http://sra.itc.it/people/cunduy/ecat

14 www.cs.rmit.edu.au/agents/pdt