

Towards Agent-Oriented Software Development

Jaelson Castro

Centro de Informática
Universidade Federal de Pernambuco
Av. Prof. Luiz Freire S/N
Recife PE, Brazil 50732-970
+1 5581 2718430
jbc@cin.ufpe.br

Manuel Kolp

Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto M5S3G4, Canada
+1 416 978 7569
mkolp@cs.toronto.edu

John Mylopoulos

Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto M5S3G4, Canada
+1 416 978 5180
jm@cs.toronto.edu

ABSTRACT

Agent-oriented computing is emerging as a powerful new paradigm that might be the cornerstone for the next generation of software like e-business systems. Naturally, defining accurate development methodologies for such emerging systems is becoming one promising area in software development. Up to now, software development techniques have been traditionally implementation-driven in the sense that the programming paradigm of the day dictated the design and requirements analysis techniques used. However, in this paper, we explore a development methodology for agent-oriented systems which is *requirements-driven*: the concepts used in upstream phases to define requirements for a software system are also used later on during downstream phases. Our proposal adopts Eric Yu's *i** [Yu95], a modeling framework for early requirements, based on the notion of *distributed intentionality* and uses it all along the life-cycle as a foundation to model late requirements, architectural and detailed design and implementation. That allows to deal with intention-based software units at the right phase and not to freeze them earlier in the process. The methodology, named *Tropos*, complements proposals for agent-oriented platforms. Since focusing on human-like and distributed behaviors, they require these kinds of behavior be modeled at a more abstract level than the current modeling approaches.

Keywords

Software development, software requirements analysis and design, agent-oriented systems, software architectures.

1 INTRODUCTION

The term agent is now widely used in software architecture to describe a range of software components, varying in capability from simple wizards in e-planning applications, to information agents used to automate information retrieval on the internet, and finally to full-fledged intelligent agents capable of reasoning in a well-defined way like those used to support ontology services in agent-based frameworks for knowledge management.

Agent-oriented computing [Sho93] has been emerging as a new technology for the next generation of software including internet-based systems. Arising from research in distributed artificial intelligence, it addresses the need for software systems to exhibit rational, human-like behavior. Traditional software systems make it difficult to model rational behavior, and often programs written in these systems experience limitations when attempting to consider such human-like behaviors, e.g., goals, intentions, plans, beliefs, desires or trusts. Surely, designing agent-oriented software systems requires proper specific development methodologies [Igl98, Jen00, Woo00].

Up to now, software development techniques have traditionally been inspired and driven by the programming paradigm of the day. This means that the concepts, methods and tools used during all phases of development were based on those offered by the pre-eminent programming paradigm. So, during the era of structured programming, structured analysis and design techniques were proposed [Dem78,You79], while object-oriented programming has given rise more recently to object-oriented design and analysis [Boo99,Wir90]. For structured development techniques this meant that throughout software development, the developer can conceptualize her software system in terms of functions and processes, inputs and outputs. For object-oriented development, on the other hand, the developer thinks throughout in terms of objects, classes, methods, inheritance and the like.

Using the same concepts to align requirements analysis with software design and implementation makes perfect sense. For one thing, such an alignment reduces impedance mismatches between different development phases. Think what it would be like to take the output of a structured analysis task, consisting of data flow and entity-relationship diagrams, and try to produce out of it an object-oriented design! Moreover, such alignment can lead to coherent toolsets and techniques for developing software (and it has!). As well, it can streamline the development process itself.

But, why base such an alignment on implementation concepts? Requirements analysis is arguably the most important stage of software development. This is the phase where technical considerations have to be balanced against social and personal ones. Not surprisingly, this is also the phase where the most and costliest errors are introduced to a software system. Even if (or rather, when) the importance of design and implementation phases wanes sometime in the future, requirements analysis will remain a critical phase for the development of any software system, answering the most fundamental of all design questions: “what is the system intended for?”

Moreover, even if the agent-oriented paradigm follows the same underlying principle as the today state-of-the-art object-oriented paradigm -- that reliable and scalable development can be enhanced by encapsulating the desired behavior in modular units which contain all the definitions and structures required for them to operate independently –agents extend this concept of encapsulation to include a representation of human-like behaviors at a higher abstraction level, above object-oriented constructs. These human-like behaviors and distributed intentions then require to be modeled all along the process at a more conceptual and declarative level than the one allowed by traditional implementation-driven methodologies. This approach leads to treat intention-based entities at the right phase in the development life-cycle and not to lose them when “operationalized” too earlier in the process. Such methodological philosophy complements agent key characteristics that make the new paradigm attractive like flexibility, suitability for distributed applications, real-time performance and decision, ability to act autonomously or to work in teams according the context.

This paper then speculates on the nature of a software development framework, named *Tropos* [Cas00,My100], which is requirements-driven in the sense that it is based on concepts used during early requirements analysis. To this end, we adopt the concepts offered by *i** [Yu95], a modeling framework offering concepts such as *actor*, *agent*, *position* and *role*, as well as social dependencies among actors, including *goal*, *softgoal*, *task* and *resource* dependencies. These concepts are used in an example to model not just early requirements for e-business management system, but also late requirements, architectural design and detailed design.

The proposed methodology spans five phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an existing organizational setting; the output of this phase is an organizational model which includes relevant actors and their respective goals.
- Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities.

- Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows.
- Detailed design, where each architectural component is defined in further detail in terms of inputs, outputs, control, and other relevant information.
- Implementation, where the actual implementation of the system is carried out, consistently with the detailed design; we use JACK, a commercial agent programming platform, based on the BDI (Beliefs-Desires-Intentions) agent architecture for this phase.

Section 2 describes a case study specification for a B2C (business to consumer) e-commerce application. Section 3 outlines our methodology. Section 4 introduces the primitive concepts offered by *i** and illustrates their use with an example. Sections 5, 6, and 7 illustrate how the technique might work for late requirements, architectural design and detailed design respectively. Throughout, we assume that the task at hand is to build generic software to support item orders processing for a media shop e-commerce application. Section 8 sketches the implementation in an intelligent agent development environment. Section 9 discusses the forms of analysis that are supported in *Tropos*. Finally, Section 10 summarizes the contributions of the paper, offers an initial self assessment of the proposed development technique, and outlines directions for further research.

2 A CASE STUDY SPECIFICATION

Media Shop is a store selling and shipping different kinds of media items such as books, newspapers, magazines, audio cds, videotapes, DVD, cdroms, games, softwares. *Media Shop* customers (on-site or remote) can use a regularly updated catalogue describing available media items to make their order. *Media Shop* is supplied with latest releases and in-catalogue items by *Media Supplier*. To increase market share, *Media Shop* has decided to open up a B2C retail sales front on the internet. With the new setup, a customer can order *Media Shop* items in person, by phone, or through the internet. The system has been named *Medi@* and is available on the word-wide-web using communication facilities provided by *Telecom Cpy*. It also uses financial services supplied by *Bank Cpy* in respect of on-line money transactions.

The basic objective for the new system is to allow an on-line customer to examine the items in the *Medi@* internet catalogue, also to place orders.

Medi@ is supposed to be available to any potential customer with internet access and a web browser. There are no registration restrictions, or identification procedures to navigate the catalogue. Even if not purchasing anything, an anonymous visitor is considered an on-line customer of *Medi@*.

Potential customers can search the on-line store by either browsing the catalogue or querying the item database. The catalogue groups media items of the same type into (sub)hierarchies and genres (e.g., audio cds decomposed into pop, rock, jazz, opera, world, classical music, soundtrack, ...) so that customers can browse only (sub)categories that interest them.

An on-line search engine allows customers with particular items in mind to search title, author/artist and description fields through keywords or full-text search. If the item is not available in the catalogue, the customer has then the possibility to ask *Media Shop* to order the desired item to *Media Supplier* provided that the customer gives mandatory editor/publisher references (e.g., ISBN, ISSN), and identifies herself (e.g., credit card number). Other internet visitors are just expected to navigate the catalogue by browsing *Medi@* offerings.

Details about media items include title, media category (e.g., book) and genre (e.g., science-fiction), author/artist, short description, editor/publisher international references and information, date, cost, and sometimes pictures (when available).

3 OUTLINE OF THE METHODOLOGY

Step 1. Acquisition of Early Requirements. The outputs of this phase are two models.

1.1 Strategic Dependency (SD) Model to capture relevant actors, their respective goals and their interdependencies.

1.2 Strategic Rationale (SR) Model to determine through a means-end analysis how the goals can be fulfilled through the contributions of other actors.

Step 2. Definition of Late Requirements in i^* . The outputs of this phase are revised SD and SR models.

2.1 Include in the original Strategic Dependency (SD) Model an actor to represent the software system to be developed.

2.2 Take this system actor and do a means-end analysis to produce a new Strategic Rationale (SR) Model.

2.3 If necessary decompose the system actor into several sub-actors and revise the SD and SR Models.

Step 3. Architectural design. The outputs of this phase are a Non Functional Requirements (NFR) Diagram and revised SD and SR models.

3.1 Select an architectural style using as criteria the desired qualities identified in Step 2. Produce a NFR diagram to represent the selection and design rationale.

3.2 If required, introduce new system actors and dependencies, as well as the decomposition of existing actors and dependencies into sub-actors and sub-dependencies. Revise the SD and SR Models.

3.3 Assigning actors to agents, positions and roles.

Step 4. Detailed design. The outputs of this phase are Class Diagrams, Sequence Diagrams, Collaboration Diagrams and Plan Diagrams.

4.1 Based on the SD and SR models produce a Class Diagram.

4.2 Develop Sequence and Collaboration diagrams to capture inter-actor dynamics,

4.3 Develop Plan (state-based) Diagrams to capture both intra-actor and inter-actor dynamics.

Step 5. Implementation. The output of this phase is a BDI (Beliefs-Desires-Intentions) agent architecture.

5.1 From the detailed design generate Agents, Capabilities, Database Relations, Events and Plans in JACK.

4 EARLY REQUIREMENTS WITH i^*

During early requirements analysis, the requirements engineer is supposed to capture and analyze the intentions of stakeholders. These are modeled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non-functional requirements of the system-to-be [Dar93]. In i^* (which stands for “distributed intentionality”), early requirements are assumed to involve social actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The i^* framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors. These models have been formalized using intentional concepts from AI, such as goal, belief, ability, and commitment (e.g., [Coh90]). The framework has been presented in detail in [Yu95] and has been related to different application areas, including requirements engineering [Yu93], business process reengineering [Yu96], and software processes [Yu94].

A strategic dependency model is a graph, where each node represents an *actor*, and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goal. We call the depending actor the *dependor* and the actor who is depended upon the *dependee*. The object around which the dependency centers is called the *dependum*. By depending on another actor for a dependum, an actor is able to achieve goals that it is otherwise unable to achieve on its own, or not as easily, or not as well. At the same time, the dependor becomes vulnerable. If the dependee fails to deliver the dependum, the dependor would be adversely affected in its ability to achieve its goals. Figure 1 shows the beginning of an *i** model consisting of two relevant actors coming from our *Media Shop* example.

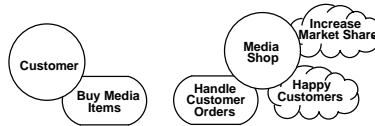


Figure 1: “Customers want to buy media items, while the *Media Shop* wants to increase market share, handle orders and keep customers happy”

The two main stakeholders for a B2C application are the customer and the business actors named respectively in our case *Media Shop* and *Customer*. The customer has one relevant goal *Buy Media Items* (represented as an oval-shaped icon), while the media store has goals *Handle Customer Orders*, *Happy Customer*, and *Increase Market Share*. Since the last two goals are not well-defined, they are represented in terms of softgoals (shown as cloudy shapes).

Once the relevant stakeholders and their goals have been identified, a strategic rationale model determines through a means-ends analysis how these goals (including softgoals) can actually be fulfilled through the contributions of other actors. A strategic rationale model is a graph with four main types of nodes -- goal, task, resource, and softgoal -- and two main types of links -- means-ends links and process decomposition links. A strategic rationale graph describes the criteria in terms of which each actor selects among alternative dependency configurations.

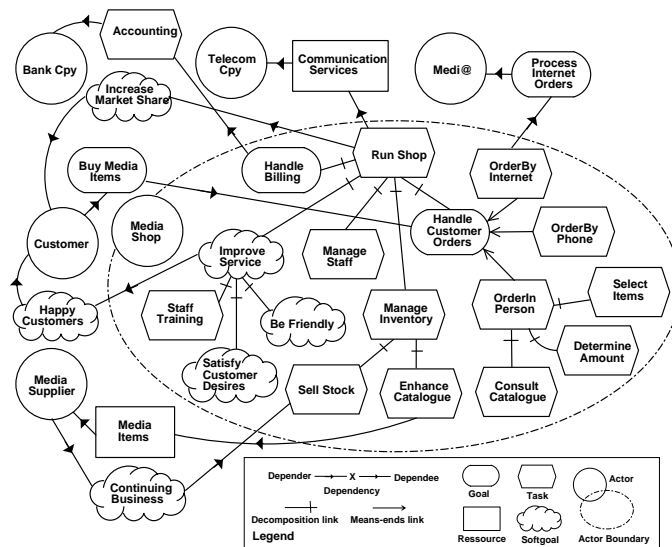


Figure 2: Means-ends analysis for the softgoal *Increase Market Share*

Figure 2 focuses on one of the (soft)goal identified for *Media Shop* namely *Increase Market Share*. The analysis is thus carried out from the perspective of the *Media Shop* actor, who has that softgoal in the first place. The analysis postulates a task *Run Shop* (represented in terms of a hexagonal icon) through which *Increase Market Share* can be fulfilled. Tasks are partially ordered sequences of steps intended to accomplish some (soft)goal. Tasks can be decomposed into goals and/or subtasks, whose collective fulfillment completes the task. In the figure, *Run Shop* is then decomposed into goals *Handle Billing* and *Handle Customer Orders*, tasks *Manage Staff* and *Manage Inventor*, and softgoal *Improve Service* which together accomplish the top-level task. In turn, sub-goals and subtasks can be refined in more precise purposes. For instance, the goal *Handle Customer Orders* is fulfilled either through tasks *OrderByPhone*, *OrderInPerson* or *OrderByInternet* while the task *Manage Staff* would be collectively accomplished by tasks *Sell Stock* and *Enhance Catalogue*. Decompositions through means-ends analysis allow us to identify actors who can accomplish a goal, carry out a task, or deliver on some needed resource. Such dependencies in Figure 2 are, among others, the resource dependency on the actor *Media Supplier* for supplying media items to enhance the catalogue, the softgoal dependencies on *Customer* for increasing market share (by running the shop) and keeping customers happy (by improving service) or the task dependency *Accounting* on *Bank Cpy* to make financial records of business transactions for *Media Shop*.

5 LATE REQUIREMENTS ANALYSIS

Late requirements analysis results in a requirements specification document which describes all functional and non-functional requirements for the system-to-be. In *Tropos*, the software system is represented as one or more actors which participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors which contribute to the fulfillment of stakeholder goals. For our example, the *Medi@* software system is introduced as an actor in the strategic dependency model depicted in Figure 3.

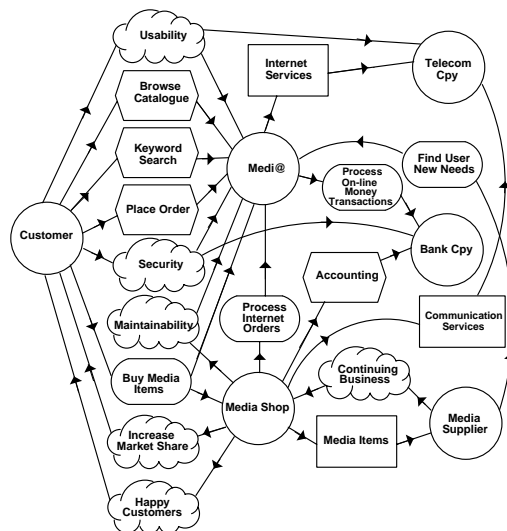


Figure 3: Strategic dependency model for a media shop

With respect to the actors identified in Figure 2, the *Customer* depends on the *Media Shop* to buy media items while the *Media Shop* depends on the *Customer* to increase market share and keep customers happy. The *Media Supplier* is expected to provide the *Media Shop* with media items because of his dependence on the latter for continuing long-term business. It can also access *Medi@* to determine new

needs from the customers, i.e., media items not available in the catalogue. As indicated earlier, the *Media Shop* depends on the *Medi@* software system for processing internet orders and on the *Bank Cpy* to take in charge business transactions. The *Customer*, in turn, depends on the *Medi@* actor to place orders through the internet, to search the database for keywords or simply to browse the on-line catalogue. On a quality software point of view, the *Customer* requires the software system as well as on-line money transaction services from the *Bank Cpy* and internet services from the *Telecom Cpy* to be respectively secure and usable. Another softgoal dependency relies *Media Shop* on *Medi@* on the way the software system should be maintainable with respect to *Media Shop* management staff's desiderata (e.g., catalogue enhancing, item database evolution, user interface update, ...). The other dependencies have already been described in Figure 2.

Although a strategic dependency model provides hints about why processes are structured in a certain way, it does not sufficiently support the process of suggesting, exploring, and evaluating alternative solutions.

As late requirements analysis proceeds, the *Medi@* software system is given additional responsibilities, and ends up as the depender of several dependencies. Moreover, the system is decomposed into several sub-actors which take on some of these responsibilities. This decomposition and new responsibilities identification is realized using the same kind of means-ends analysis along with the strategic rationale analysis illustrated in Figure 2.

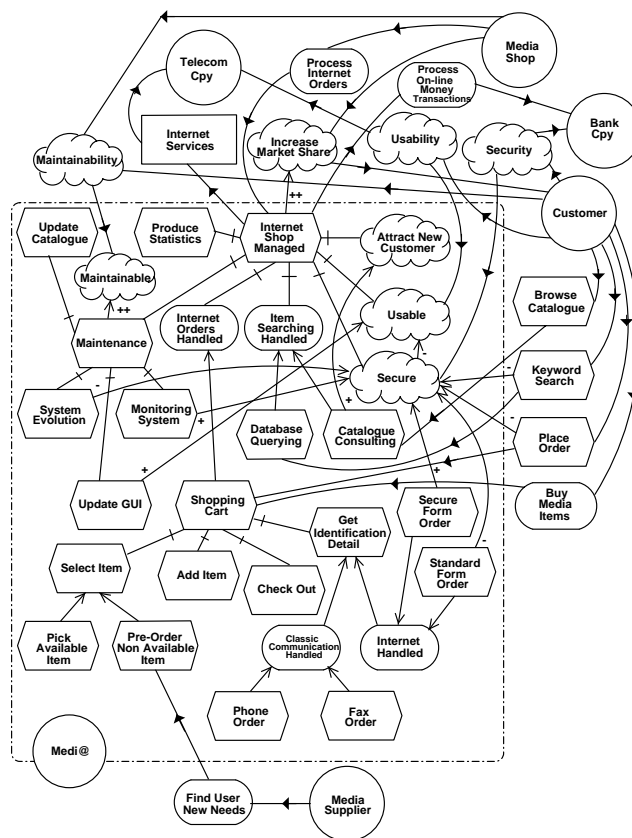


Figure 4: Strategic rationale model for the *Medi@* system actor

Hence, the analysis in Figure 4 focuses this time on the system software *Medi@*. It postulates a root task *Internet Shop Managed* providing sufficient support (++) [Chu00] to the softgoal *Increase Market Share*,

associated during early requirement analysis with the *Media Shop* actor. That top-level task is firstly refined into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Secure* and *Usable* and tasks *Produce Statistics* and *Maintenance*. They together realize the top-level task, each of them focusing on one main software responsibility.

To manage internet order, *Internet Order Handled* is achieved through the task *Shopping Cart* which is, in turn, decomposed into subtasks *Select Item* and *Check Out*, and *Get Identification Detail*, each of them covering one of the main process activities required to design an operational on-line shopping cart [Con00]. The latter goal is achieved alternatively through sub-goal *Classic Communication Handled* dealing with phone and fax orders or sub-goal *Internet Handled* managing secure form or standard form orderings. To allow ordering new items, not listed in the catalogue, *Select Item* is also further refined into two alternative subtasks, one dedicated to selecting catalogued items, the other to preordering non available products.

To provide sufficient support (++) to the *Maintainability* softgoal, *Maintenance* is refined into three subtasks dealing with catalogue updating, system evolution and system monitoring.

The goal *Item Searching Handled* might alternatively be fulfilled through tasks *Database Querying* or *Catalogue Consulting* with respect to customers' navigating desiderata, i.e., searching precisely particular items in mind by using search functions or simply browsing the catalogued products.

In addition, as already pointed, Figure 4 introduces softgoal contributions to model sufficient or partial positive (respectively ++ and +) or negative (respectively - - and -) support to softgoals *Secure*, *Usable*, *Maintainable*, *Attract New Customers* and *Increase Market Share*. As will be explained in Section 5, this kind of softgoal decomposition will be fully used to model, at the architectural design level, non-functional requirements captured by the *Security*, *Usability* and *Maintainability* softgoals.

The result of this means-ends analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been generated.

Figure 5 suggests one possible assignment of responsibilities identified from the *Medi@* strategic rational model. The *Medi@* system is decomposed into four sub-actors: *Store Front*, *Billing Processor*, *Service Quality Manager* and *Back Store*.

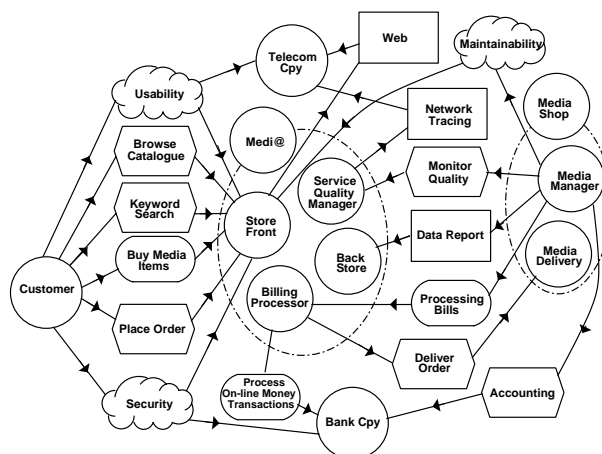


Figure 5: The web system consists of four inside actors, each with external dependencies

Store Front principally interacts with the *Customer* actor and provides her with a usable front-end web application. *Back Store* keeps track of all web information about customers, products, sales, bills and other data of strategic importance to *Media Shop*. *Billing Processor* is in charge of the secure management of orders and bills, and other financial data; also of interactions to *Bank Cpy*. *Service Quality Manager* is introduced in order to look for *security* gaps, *usability* bottlenecks and *maintainability* issues.

All four sub-actors need to communicate and collaborate in running the system. For instance, *Store Front* communicates to *Billing Processor* relevant customer information required to process bills. *Store Front* and *Billing Process* are supervised by *Service Quality Manager* who monitors transactions. *Back Store* organizes, stores and backs up all information coming from *Store Front* and *Billing Processor* in order to produce statistical analyses, historical charts and marketing data. A further refinement of each of these software system sub-actors with new dependencies and responsibilities will be done in Section 5 at the architectural design level.

For the rest of the section, we focus on *Store Front*. This actor is in charge of catalogue browsing and item database searching, also provides on-line customers with detailed information about media items. We assume that different media shops working with *Medi@* may want to provide their customers with various forms of information retrieval (boolean, keyword, thesaurus, lexicon, full text, indexed list, simple browsing, hypertext browsing, SQL queries, etc.).

Store Front is also responsible for supplying a customer with a web shopping cart to keep track of items the customer is buying when visiting *Medi@*. We assume that different media shops using the *Medi@* system may want to provide customers with different kinds of shopping carts with respect to their internet browser, plug-ins configuration or platform or simply personal wishes (e.g., Java mode shopping cart, simple browser shopping cart, frame-based shopping cart, CGI shopping cart, enhanced CGI shopping cart, shockwave-based shopping cart,...)

Finally, *Store Front* also initializes the kind of processing that will be done (by *Billing Processor*) for a given order (phone/fax, internet standard form or secure encrypted form). We assume that different media shop managers using the *Medi@* web system may be processing various types of orders, such as those listed above differently and that customers may be selecting the kind of delivery system they would like to use (UPS, FedEx, DHL, express mail, normal mail, overseas service, ...).

Resource, task and softgoal dependencies correspond naturally to functional and non-functional requirements. Leaving (some) goal dependencies between system actors and other actors is a novelty. Traditionally, functional goals are “operationalized” during late requirements [Dar93], while quality softgoals are either operationalized or “metricized” [Dav93]. For example, *Billing Processor* may be operationalized during late requirements analysis into particular business processes for processing bills and orders. Likewise, a security softgoal might be operationalized by defining interfaces which minimize input/output between the system and its environment, or by limiting access to sensitive information. Alternatively, the security requirement may be metricized into something like “No more than X unauthorized operations in the system-to-be per year”.

Leaving goal dependencies with system actors as dependees makes sense whenever there is a foreseeable need for flexibility in the performance of a task on the part of the system. For example, consider a communication goal “communicate X to Y”. According to conventional software development techniques, such a goal needs to be operationalized before the end of late requirements analysis, perhaps into some sort of a user interface through which user Y will receive message X from the system. The problem with this approach is that the steps through which this goal is to be fulfilled (along with a host of background assumptions) are frozen into the requirements of the system-to-be. This early translation of goals into concrete plans for their fulfillment makes software systems fragile and less reusable.

In our example, we have left three goals in the late requirements model. The first goal is *Usability*

because we propose to implement *Store Front* and *Service Quality Manager* as agents able to automatically decide at run-time which catalogue browser, shopping cart and order processor architecture fit better to the customer's needs or navigator/platform specifications. Moreover, we would like to include different kinds of search engine reflecting search techniques proposed in information brokering or retrieval and let the system dynamically chooses the most appropriate with respect to the customer's needs. The second goal in the late requirements specification is *Security*. To fulfil it, we propose to provide in the system's architecture a number of security strategies and let the system decide at run-time which one is the most appropriate, taking into account environment configurations, web browser specifications and network protocols used. The third goal is *Maintainability* since we would like to let the software system -- not only the catalogue content but also the database schema or architectural model, the server structure and behaviors and the application functionalities -- be dynamically extended to integrate new and future web related technologies.

Hence, instead of operationalizing these goals during requirements analysis, we propose to do so during architectural design.

6 ARCHITECTURAL DESIGN

The architectural design has emerged as a crucial phase of the design process consisting of a number of structural elements and their interfaces. A software architecture constitutes a relatively small, intellectually manageable model of system structure, and how system components work together. For our internet counter example, the task is to define (or choose) a web application architecture. The canonical web architecture consists of a web server, a network connection, HTML/XML documents on one or more clients communicating with a Web server via HTTP, and an application server which enables the system to manage business logic and state (see Figure 6). This architecture is not intended to imply that a web application cannot use distributed objects or Java applets; nor does it imply that the web server and application server cannot be located on the same machine.

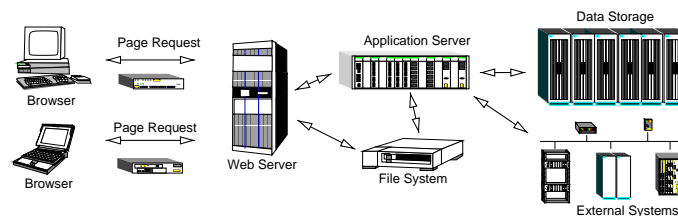


Figure 6: Canonical Web Architecture

Indeed, software architects have developed catalogues of web architectural style (see, for example, [Con00]). The three most common styles are the *Thin Web Client*, *Thick Web Client* and *Web Delivery*. The *Thin Web Client* is most appropriate for internet-based web applications, in which the client has minimal computing power or no control over its configuration. The client requires only a standard forms-capable web browser. All the business logic is executed on the server during the fulfillment of page requests for the client browser. The *Thick Web Client* style extends the *Thin Web Client* style with the use of client-side scripting and custom objects, such as ActiveX controls and Java applets. A significant amount of business logic can be executed on the client machine. Finally, in the *Web Delivery* style, the web is used primarily as a delivery mechanism for an otherwise traditional client/server system. The client communicates directly with object servers, bypassing HTTP. This style is appropriate when there is significant control over client and network configuration.

During architectural design we concentrate on the key system actors, defined during late requirements analysis, and their responsibilities. These include the desired functionality of the system-to-be, as well as a number of quality requirements related to usability, security, performance, portability, availability, reusability, comprehensibility, evolvability, extensibility, modularity, reusability, etc.

Functional requirements can be accommodated using one of several standard methodologies, such as structured analysis and design, or object-oriented design methods. However, quality requirements are generally not addressed by such techniques [Chu00]. For example, as we are building an Internet application, security is certainly an important concern. Indeed, this was captured by the *Security* software goal dependency between the *Customer* and *Medi@* actors (see Figure 3). The software application should do only what it is supposed to do, without compromising the integrity of the data by exposing them to unauthorised users. Likewise, *Usability* is a concern, since customers may have little internet experience. Interfaces need to be carefully crafted to handle in a user-friendly and comprehensible manner the communication between the customer and the system, as well as the flow of activities of the business process. To deal with this softgoal, we have introduced a *Usability* softgoal dependency between *Customer* and *Medi@* actors (see Figure 3). Similarly, *Maintainability* is a strategic design issue since the software application should be able to integrate new kinds of server modules, languages or internet protocols in a flexible and generic manner without having to redesign the whole application from scratch.

To cope with these goals, the software architect, who is another (external) actor, goes through a means-ends analysis comparable to what was discussed earlier. In this case, the analysis involves refining the softgoals to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them, as shown in Figure 7. This analysis is intended to make explicit the space of alternatives for fulfilling the top-level quality softgoals. Moreover, the analysis allows the evaluation of several alternative architectural styles. The styles are represented as operationalized softgoals (saying, roughly, “make the architecture of the new system *Web Delivery-/Thin Web-/Thick Web-based*”) and are evaluated with respect to the alternative non-functional softgoals as shown in Figure 6. The evaluation results in contribution relationships from the architectural goals to the quality softgoals, labeled “+”, “++”, “-”, “--”, to model partial/sufficient positive/negative contributions as already explained. Design rationale is represented by claim softgoals drawn as dashed clouds. They make it possible for domain characteristics (such as priorities) to be considered and properly reflected into the decision making process, e.g., to provide reasons for selecting or rejecting possible solutions (+, -). Exclamation marks (! and !!) are used to mark priority softgoals while a check-mark “✓” indicates an accepted softgoal and a cross “✗” labels a denied softgoal.

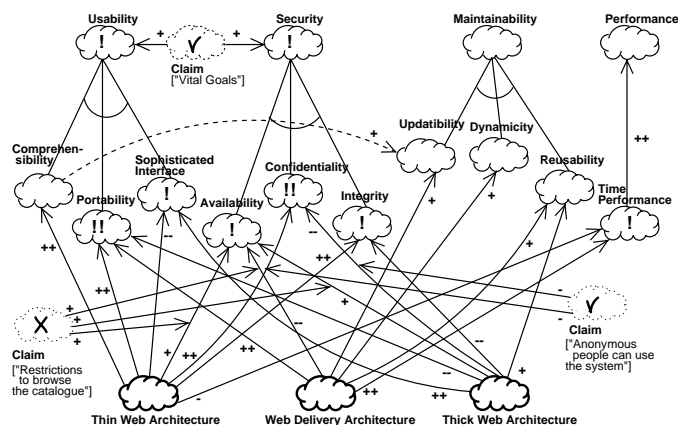


Figure 7: Refining softgoals in architectural design

The *Usability* softgoal has been AND-decomposed into sub-goals *Comprehensibility*, *Portability* and *Sophisticated Interface*. From the customer point of view it is important for the *Medi@* application to be comprehensible at first glimpse, i.e., intuitive and ergonomic. The look and feel of the interface must naturally guide the customer's actions who is required only to have minimal computer knowledge. Equally strategic to *Usability* is the portability of the application across browser implementations and the quality of the interface. Note that not all HTML browsers support scripting, applets, controls and plug-ins. These technologies make the client itself more dynamic, and capable of animation, fly-over help, and sophisticated input controls. When only minimal business logic needs to be run on the client, scripting is often an easy and powerful mechanism to use. When truly sophisticated logic needs to run on the client, building Java applets, Java beans, or ActiveX controls is probably a better approach. ActiveX, however, is an option only when the client computers are Windows-based.

The *Security* softgoal has initially been AND-decomposed into sub-goals *Availability*, client *Confidentiality* and data *Integrity*. The first softgoal guards against interruption of service, the second guards against unauthorized disclosure of information and the third checks the completeness and the accuracy of data transactions. Network communication may not be very reliable causing sporadic loss of the server. Clients, especially those on the internet are, like servers, at risk in web applications. It is possible for web browsers to unknowingly download content and programs that could open up the client system to crackers and automated agents all over the net. JavaScript, Java applets, ActiveX controls, and plug-ins all represent a certain degree of risk to the client and the information it manages.

The *Maintainability* softgoal has been AND-decomposed into subgoals *Reusability*, *Updatability* and *Dynamicity*. The first softgoal deals with the way to reuse software components. Even if we can assume that there is a certain uniformity in the client side mainly because Microsoft Windows is a predominant operating system for desktops, server components are inherently more diverse according to each web architecture we have described above. Nevertheless, they have certain elements in common, and some of those elements can be captured (e.g., by component models such as Enterprise JavaBeans or COM) to be reused. For example, almost all server-side processes of *Medi@* must be able to handle transactions. An Enterprise JavaBeans model could allow to provide components that are transaction-aware to facilitate and reuse this common task.

Updatability is a classical software quality inherent to any kind of business application involving a product database. It is however strategically important for the viability of the application, the stock management and the business itself. It then must be *comprehensible* (implicit contribution from *Comprehensibility* to *Updatability* in Figure 7) by the *Media Shop* employees having to very regularly (daily, weekly) bring up to date the catalogue by themselves for stock management consistency.

Dynamicity deals with the way the software system can be designed using generic mechanisms to allow web pages and look to be dynamically and easily changed. Indeed, information content and layout need to be frequently refreshed to give correct information to customers or simply be fashionable for marketing reasons (softgoal *Attract New Customers* in Figure 4). Frameworks like Active Server Pages (ASP), Server Side Includes (SSI) to create dynamic pages or tools simply separating generic layout from content such as GenPage to provide "smarter" (non-static) pages make this softgoal easier to achieve.

Finally, *Performance* is concerned with the capability of *Medi@* to do what needs to be done, as quickly and efficiently as possible. In particular, *Time Performance* deals with the ability of *Medi@* to respond in time to client requests for its services. Indeed, given that network latency -- the delay inherent in moving requests from clients to servers and their concomitant responses from servers to clients -- can be quite long on a network of global proportions like the Internet, adequate or accurate time performances and measures can be difficult to produce.

As shown by Figure 7, each of the three web architectural patterns contributes positively or negatively to each softgoals we have just explained. Due to the lack of space, we only described some of these contributions.

The *Thin Web Client* architecture is useful for internet-based applications, for which only the most basic client configuration can be guaranteed. Hence, this architecture does well for *Portability*. However, it has a limited ability to support *Sophisticated User Interfaces*. The browser acts as the entire user interface delivery mechanism and in most common browsers these are limited to a few text entry fields and button. Moreover, this architecture relies on a connectionless protocol such as HTTP, which contributes positively to availability of the system since the sporadic loss of a server might not pose a serious problem. Pure HTTP, without client-side scripting, is rather secure.

On the other hand, the *Thick Web Client* architecture is generally not portable across browser implementations. Not all HTML browsers support JavaScript or VBScript. Additionally, only Microsoft Windows base clients can use ActiveX controls. However, these technologies contribute very positively to the goal of having sophisticated interfaces. As in the *Thin Web Client* architecture, all communication between client and server is done with HTTP. Since HTTP is a “connectionless” type of protocol, most of the time there is no open connection between client and server. Only during page requests does the client send information. Hence its positive contribution to *Availability*. On the negative side, client-side scripting and custom objects, such as ActiveX controls and Java applets may pose risks to the client confidentiality.

Last but not least, the *Web Delivery* architecture is highly portable, since the browser has some built-in capabilities to automatically download the needed components from the server. However, this architecture requires a reliable network. Connections between client and server objects last much longer than do HTTP connections, and so sporadic loss of the server, poses a serious problem that has to be addressed for this architecture.

As with late requirements, an interesting feature of the proposed analysis method is that it is goal-oriented. Goals are introduced and analyzed during architectural design, and guide the design process.

Apart from goal analysis, this phase involves, on the one hand, the introduction of other new system actors and dependencies and on the other hand the decomposition of existing actors and dependencies into sub-actors and sub-dependencies which will assume the responsibilities of the key system actors introduced earlier, refine them or capture new ones.

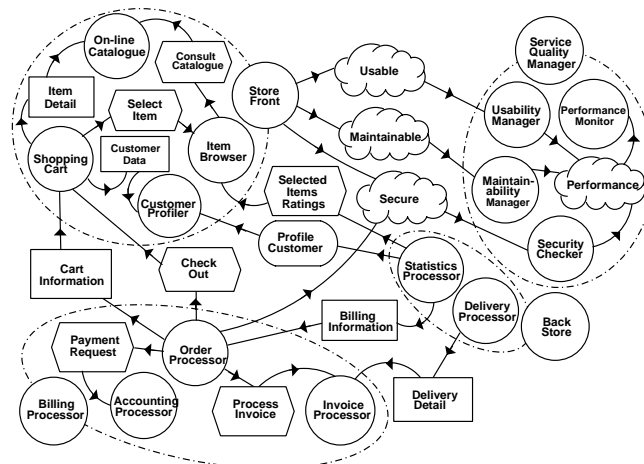


Figure 8: Strategic Dependency Model of *Medi@* system sub-actors

Figure 8 focuses on the latter kind of refinement. To accommodate the responsibilities of the *Store Front* actor of Figure 6, the architect introduces sub-actors like *Item Browser* managing catalogue navigation et item database search engines, *Shopping Cart* for selecting and customizing items, *Order Processor* for placing and tracking orders, *Customer Profiler* for tracking customer data and producing client profile

types and *On-line Catalogue* dealing with digital library obligations for example, notification of the arrival of new items and recommendation (prediction based on profile and “business intelligence”, possibly derived through data-mining techniques). Based on the non-functional requirement decomposition proposed in Figure 7, *Service Quality Manager* is further refined into four new system sub-actors *Usability Manager*, *Security Checker*, *Maintainability Processor* and *Performance Monitor*, each of them assuming one of the top main softgoals explained previously. *Billing Processor* is decomposed into *Order Processor* mainly dialoguing with *Shopping Cart* to resume ordering items and delegating invoicing to *Invoice Processor* and financial duties to *Accounting Processor* interacting with *Bank Cpy* (not represented in Figure 8). Finally, *Back Store* is refined into *Statistics Processor* concerned with duties like producing charts, reports, audits, sales, forecast turnover, and *Delivery Processor* taking care of responsibilities such as interactions with delivery companies information systems.

An interesting decision that comes up during architectural design is whether fulfillment of an actor’s obligations will be accomplished through assistance from other actors, through delegation (“outsourcing”), or through decomposition of the actor into component actors. Going back to our running example, the introduction of other actors described in the previous paragraph amounts to a form of delegation in the sense that *Store Front* retains its obligations, but delegates subtasks, sub-goals etc. to other actors. An alternative architectural design would have *Store Front* outsourcing some of its responsibilities to some other actors, so that *Store Front* removes itself from the critical path of obligation fulfilment. Lastly, *StoreFront* may be refined into an aggregate of actors which, by design, work together to fulfil *StoreFront*’s obligations. This is analogous to a committee being refined into a collection of members who collectively fulfil the committee’s mandate. It is not clear, at this point, how the three alternatives compare, nor what are their respective strengths and weaknesses.

7 DETAILED DESIGN

The detailed design phase is intended to introduce additional detail for each architectural component of a software system. In our case, this includes actor communication and actor behavior. To support this phase, we may be adopting agent communication languages, message transportation mechanisms, ontology communication, agent interaction protocols, plan model, etc. from the agent programming community. One possibility, among admittedly many, is to adopt one of the extensions to UML proposed by the FIPA (Foundation for Intelligent Agents) and the OMG Agent Work group [Bau99,Ode99,Ode00]. The rest of the section concentrates on the *Shopping cart* actor and the *check out* dependency. Figure 9 depict a partial UML class diagram focusing on that actor that will be implemented as an aggregation of several *CartForms* and *ItemLines*. Associations *ItemDetail* to *On-line Catalogue*, aggregation of *MediaItems*, and *CustomerDetail* to *CustomerProfiler*, aggregation of *CustomerProfileCards* are directly derived from resource dependencies with the same name in Figure 8.

As will be explained in the next section, our target implementation model is the BDI model [Bra87], a rational agent model whose main concepts are *Beliefs*, *Desires* and *Intentions*. According to Figure 12, we will implement *i** tasks as BDI intentions (or plans). They are represented after methods (see Figure 9) following the label “Plans:”

One of these plans is the *checkout* task dependency between *Shopping Cart* and *Order Processor* which involves a detailed design of an *agent interaction protocol* (AIP) to describe inter-agent dynamics. To define such a protocol, we use AUMML - the Agent Unified Modeling Language [Bau99], which supports templates and packages to represent the protocol as an object, but also in terms of sequence and collaborations diagrams.

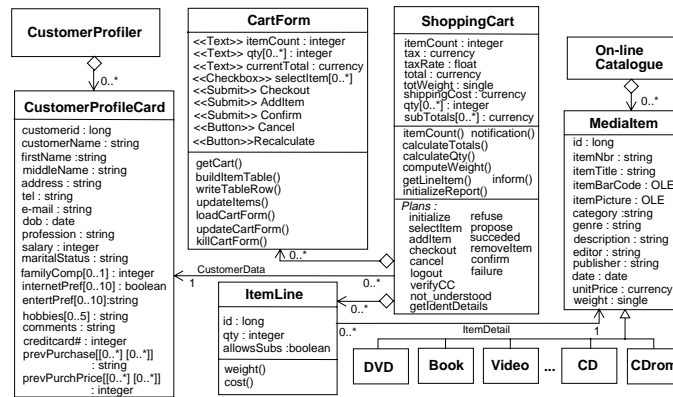


Figure 9: Partial Class Diagram of *Store Front* focusing on *Shopping Cart*

Figure 10 (a) introduces the interaction context. It depicts a general view of the sequence diagram to order media items. It is triggered by the *checkout* communication act (CA) from *Customer* to *Shopping Cart* and ended with a returned information status about the complete sequence. When the *Customer* pushes the *checkout* button, the *Shopping Cart* asks the *Order Processor* to process orders. In turn, the latter sends a *payment request* CA to *Accounting Processor* which *informs* him about the status (failure/success) of its internal processing. In the case of success, *Order Processor* concurrently asks *Invoice Processor* to process an invoice (which subsumes, a *delivery detail* CA to *Delivery Processor*) and sends *billing information* to *Statistics Processor*.

The sequence diagram of Figure 10 (a) only provides basic specification for an intra-agent order processing protocol. More processing details are required. For instance, in Figure 10 (a), the diagram stipulates neither the procedure used by the *Customer* to produce the *checkout* CA, nor the procedure employed by the *Shopping Cart* to respond to the CA.

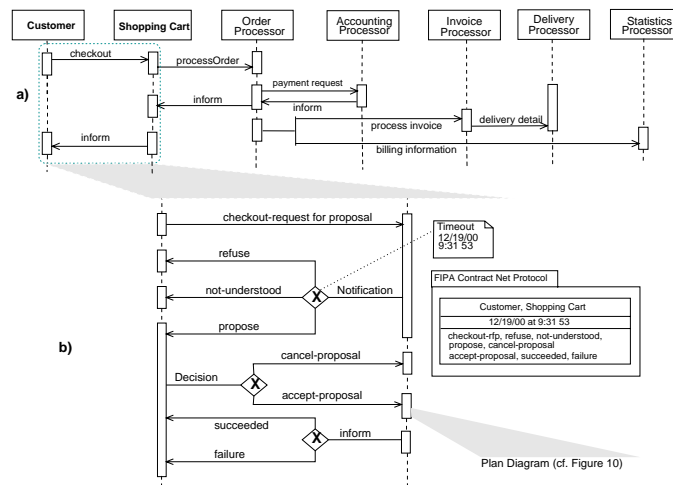


Figure 10: Sequence diagram to order media items (a) and agent interaction protocol focusing on a *checkout* dialogue (b)

As shown by Figure 10 (b), such details can be provided by using *levelling* [Ode00], i.e., by introducing additional interaction and other diagrams which describe some of the primitive action shown in Figure 10 (a). Each additional level can express *inter-actor* or *intra-actor* dialogues. At the lowest level,

specifications of an actor protocol requires spelling out the detailed processing that takes place within an actor, as will be shown in Figure 11, in order to implement the protocol.

Figure 10 (b) focuses on the AIP dialogue between *Customer* and *Shopping Cart*. It depicts a customization of the FIPA Contract Net protocol [Ode99] to that particular interaction. Such a protocol describes a communication pattern among actors as an allowed sequence of messages, as well as constraints on the contents of those messages.

When a *Customer* wants to check out, a request for proposal message (*checkout-request-for-proposal*) is sent to the *Shopping Cart*. The *Shopping Cart* actor has then to respond to the *Customer* before a given timeout (for network security and integrity reasons): by refusing to provide a proposal, submitting a proposal, or informing that it did not understand (the diamond symbol indicates a decision that can result in zero or more communications being sent – depending on the conditions it contains; the “X” in the decision diamond indicates an *exclusive or* decision). If a proposal is offered, the *Customer* has a choice of either accepting or canceling the proposal. In the case of proposal acceptance, the *Shopping Cart* actor will finally inform the *Customer* about the proposal’s execution (success/failure). The internal processing of *Shopping Cart*’s *checkout* plan is described in Figure 11.

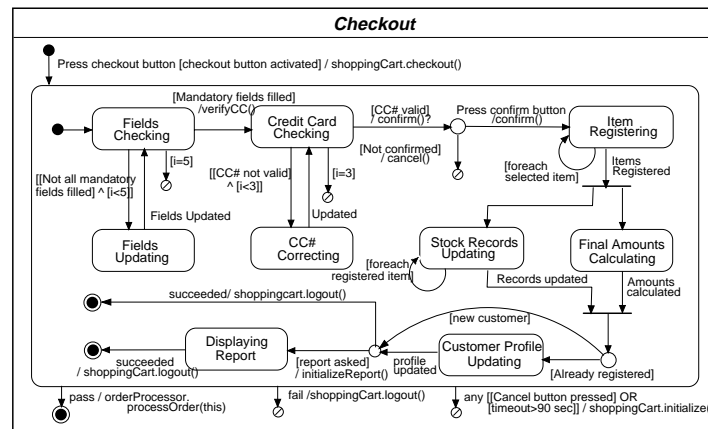


Figure 11: A plan diagram for checking out

At the lowest level, state charts and activity diagrams can be used to specify the internal processing (plans) of actors who are no more aggregates. We use them as plan diagrams as proposed in [Kin96,Kin96a]. Each identified plan is specified as a plan diagram, which is denoted by a rectangular box. The lower section, the plan graph, is a state transition diagram. However, plan graphs are not just descriptions of system behavior developed during analysis. Rather, they are directly executable prescriptions of how a BDI agent should behave (execute identified plans) to achieve a goal or respond to an event.

The elements of the plan graph are three types of node; start states, end states and internal states, and one type of directed edge; transitions. Start states are denoted by small filled circles. End states may be pass or fail states, denoted respectively by a small target or a small no entry sign. Internal states may be passive or active. Passive states have no substructure and are denoted by a small open circle. Active states have an associated activity and are denoted by rectangular boxes with rounded corners (e.g., *Fields Checking*, *Credit Card Checking*, *Item Registering*, ...). Activities may be iteration constructs, including *do* and *while* loops, or in the case of a graph state, an embedded graph called a sub-graph.

The initial transition of the plan graph is labeled with an activation event (*Press checkout button*) and activation condition (*[checkout button activated]*) which determine when and in what context the plan

should be activated. Transitions from a state automatically occur when exiting the state and no event is associated (e.g., when exiting *Fields Checking*) or when the associated event occurs (e.g., *Press cancel button*), provided in all cases that the associated condition is true (e.g., *[Mandatory fields filled]*). When the transition occurs any associated action is performed (e.g., *verifyCC()*).

Plan graphs have a semantics which incorporates a notion of failure. Failure within a graph can occur when an action upon a transition fails, when an explicit transition to a fail state occurs, or when the activity of an active state terminates in failure and no outgoing transition is enabled.

If the failure occurs inside a state, then the activity of that state terminates in failure. If the failure occurs in a plan graph, then the plan terminates in failure. If the plan has been activated to perform a goal, this may result in that goal fails, depending on the availability of alternative plans to achieve the goal.

Figure 11 depicts the plan diagram for *checkout*. It is triggered by pushing the checkout button. Mandatory fields are first checked. If all mandatory fields are not filled, an iteration allows the customer to update them. For security reasons, the loop exits after 5 tries ($[i < 5]$) and causes the plan to fail. Credit Card validity is then checked. Again for security reasons, when not valid, the CC# can only be corrected 3 times. Otherwise, the plan terminates in failure. The customer is then asked to confirm the CC# to allow item registration. If the CC# is not confirmed, the plan fails. Otherwise, each item is iteratively registered. When all items are registered, the stock records (for each registered item) are updated and the final amounts calculated concurrently. If the customer was already registered by the *customer profiler*, his profile card is updated, otherwise nothing is done. If a report is asked, the system displays a printable voucher including all the details of the checkout process. Finally, the whole plan succeeds, the *ShoppingCart* automatically logs out and asks the *Order Processor* to initialize the order. When, for any reason, the plan fails, the *ShoppingCart* automatically logs out. At anytime, if the cancel button is pressed or the timeout is > 90 sec., (e.g., due to a network bottleneck), the plan fails and the *Shopping Cart* is reinitialized.

8 MAPPING TO BDI AGENTS

JACK Intelligent Agents [Cob00] is an agent-oriented development environment built on top of the Java programming language. JACK's integration with Java is analogous to the relationship between the C++ and C languages. C was developed as a procedural language and subsequently C++ was developed to provide C programmers with object-oriented extensions. Similarly, the JACK Agent Language has been developed to provide agent-oriented specific extensions to Java. Technically, to be executed on the Java virtual machine, JACK source code is first compiled into regular Java by the JACK Agent Compiler.

Agents in JACK are said intelligent in the sense they model reasoning behavior according to the theoretical *Belief Desire Intention* (BDI) model [Bra87] used in artificial intelligence as well as in cognitive science and philosophy. Following this model (see Figure 12), JACK agents can be considered autonomous software components that have explicit *goals* to achieve or *events* to handle (desires). To describe how they should go about achieving these desires, these agents are programmed with a set of plans (intentions). Each plan describes how to achieve a goal under varying circumstances. Set to work, the agent pursues its given goals (desires), adopting the appropriate plans (intentions) according to its current set of data (beliefs) about the state of the world.

To support the BDI agents, JACK proposes five main language functional constructs. As shown in Figure 12, they are:

- *Agents* to define the behavior of intelligent software agents. They have methods and data members like objects but also capabilities, database relations, description of events and plans.

- *Capabilities* to encapsulate and aggregate events, plans, databases or other capabilities. They provide agents with a number of “capabilities”, each of which has a specific function attributed to it.
- *Database relations* to store beliefs and data that an agent has acquired.
- *Events* to identify the circumstances and messages that an agent can respond to.
- *Plans* that are instructions an agent follows to try to achieve its goals and executes to handle its designated events.

Figure 12 depicts how the i^* concepts can be mapped into the BDI model and JACK constructs and how each concept is related to the others inside the same model. i^* actors, (informational/data) resources, softgoals, goals and tasks are respectively translated into BDI agents, belief, desires and intentions. In turn, a BDI agent will be mapped as a JACK agent, a belief will be asserted (or retracted) as a database relation, a desire will be posted (sent internally) as a BDIGoalEvent (representing an objective that an agent wishes to achieve) and handled as a plan and an intention will be implemented as a plan. Finally, a i^* dependency will be directly realized as a BDIMessageEvent (received by agents from other agents).

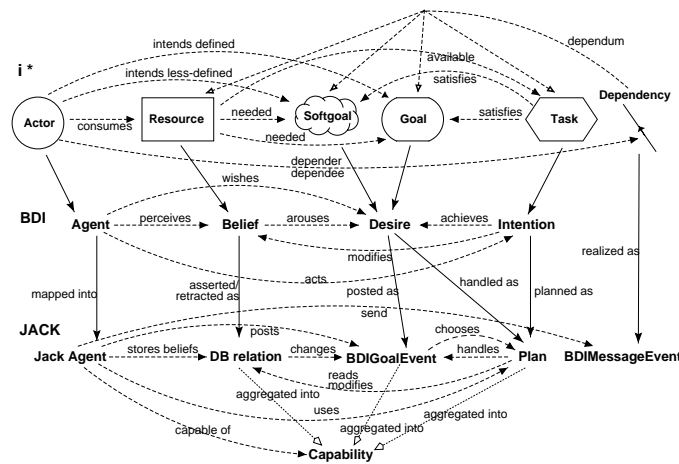


Figure 12: $i^*/BDI/JACK$ mapping overview

As illustrated in Section 6, other models and diagrams must also be used in the detailed design phase to model additional system details not captured by i^* . For instance, Figure 10 (b) represented the AIP dialogue between *Customer* and *Shopping Cart* involving a *checkout-request-for proposal*. Figure 13 depicts the JACK layout presenting each of the five JACK constructs as well as the implementation of the first part of Figure 10 (b) dialogue. *Customer* and *Shopping Cart* are implemented as JACK agents (*extends Agent*). The request for proposal *checkout-rfp* is a MessageEvent (*extends MessageEvent*) sent by *Customer* and handled by the *Shopping Cart*'s *checkout* plan (*extends Plan*) we have detailed in Figure 11.

In response to *checkout-rfp*, *Shopping Cart* posts a *notification* MessageEvent handled by (one of the) three plans *refuse*, *propose*, *not-understood*. Finally, *Timeout* (which we consider a belief) is implemented as a closed world (i.e., true or false) database relation asserting for each *Shopping Cart* one or several timeout delays.

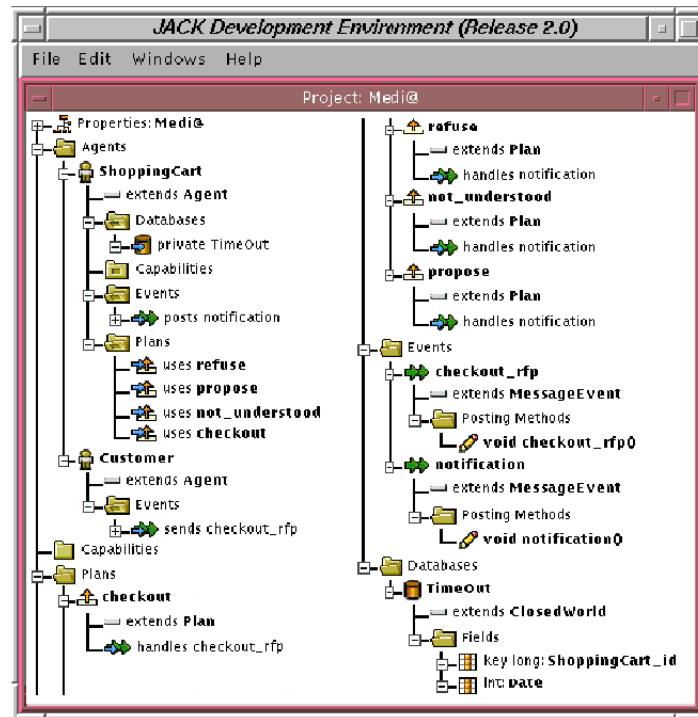


Figure 13: Partial implementation of Figure 9 in JACK

9 FORMS OF ANALYSIS IN TROPOS

To supplement diagrams with rigorous definitions of the actors, dependencies and relevant entities and relationships, we adopt a KAOS-like [Dar93] notation. Figure 14 represents some of the definitions coming from Figure 3. We focus on the actors *Customer* and *Media Shop* and their dependencies to each other (*BuyMediaItems* and *IncreaseMarketShare*).

Entity Order

Has orderId: Long, cust: Customer, orderDate: Date,
items: **SetOf** [mediaItem]

Invariant $(\forall x) (\text{Order}(x) \wedge \bullet \neg \text{Order}(x) \Rightarrow \diamond \text{OrderItemsOK}(x.\text{items}))$
End Order

Entity MediaItem

Has itemId: Long, itemTitle: String, description: Text, editor: String ...
End MediaItem

Action MakeOrder

Input Customer {**Arg**: cust}, Date {**Arg**: orderDate}, **SetOf** [mediaItem] {**Arg**: items}

Output Order [**Arg**: order]

Precondition $\neg \text{OrderItemsOK}(\text{order})$

Postcondition $\text{order.cust} = \text{cust} \wedge \text{order.orderDate} = \text{orderDate} \wedge \text{order.items} \subseteq \text{items}$

End MakeOrder

Actor Customer

Has customerId: Long, customerName: Name, address: Address,
tel: PhoneNumber, ...

Capable of MakeOrder, Pay, Browse, Query, ...

End Customer

```

Actor MediaShop
Has name: {MediaLive}, address: {"735 Yonge Street"},
      phone#: 0461-762-883
      Capable of Sell, Ship, SendInvoice, ...
End MediaShop

Relationship OrderedBy
Links Customer {Role ordering, Card 0..*}
      MediaItem {Role ordered, Card 0..*}
End OrderedBy

Dependency BuyMediaItems
Type Goal
Mode Achieve
DefinedAs OrderItemsOK(order)
Depender Customer {Card 0..*}
Dependee MediaShop {Card 0..*}
Has order: Order
Invariant (  $\forall o: \text{Order}$  ) (Fulfil (the BuyMediaItems(o))
       $\Rightarrow \diamond \text{Fulfill}(\text{the PlaceOrder}(o))$ )
Invariant ( $\forall o$ ) ( $\text{Order}(o) \wedge \bullet \neg \text{Order}(o) \Rightarrow \diamond (\exists d: \text{BuyMediaItems},$ 
      ms: MediaShop) ( $o = d.\text{order} \wedge \text{bm.ordering} = d.\text{Depender} \wedge$ 
      ms = d.Dependee)
End BuyMediaItems

Dependency IncreaseMarketShare
Type Softgoal
      Mode Maintain
Depender MediaShop
Dependee Customer
Key (dependee, depender)
Invariant ( $\forall \text{ms}: \text{MediaShop}, c: \text{Customer}$ )
      (Maintain (the IncreaseMarketShare(c,ms))  $\Rightarrow$ 
      ( $\blacklozenge \text{Fulfill}(\text{the BuyMediaItems}(\text{order})) \Rightarrow$ 
       $\blacklozenge \text{Fulfill}(\text{the PlaceOrder}(\text{order}))$ ))
End IncreaseMarketShare

```

Figure 14: KAOS-like specifications for Figure 3

For the specification language and the notation used in Figure 14, we assume that for every class there is a fluent, e.g., *Order(.)*. A referential expression has one of three forms: variable or constant; *expr.attr*, where *attr* is an attribute of the value of *expr*. In *Class(key)*, *Class* is a class and *key* one of its keys. We use a Temporal Logic, à la KAOS, where *P* is asserted with respect to current time: $\bullet P$ and $\circ P$ assert *P* with respect to the previous/next time instance while $\blacklozenge P$ and $\diamond P$ assert *P* for sometime in the past/future. We use different actions for dependency fulfilment: e.g., *Fulfill* is a one time fulfillment of a dependency by the dependee; *Maintain* is a continuing fulfilment.

Another high level language used in *Tropos* is a ConGolog –like language [Les99] allowing us to specify dynamics such as processes modeled by plan diagrams (see Figure 11). Primitive actions can be defined in terms of pre- and post-conditions and decomposed into procedures using modeling constructs like sequencing ($a_1 ; a_2$), conditional (*if-then*), iteration (*while <condition> do*), concurrent activities ($a_1 \parallel a_2$), priority ($a_1 \gg a_2$), non-deterministic choice ($a_1 \mid a_2$), interrupt ($\langle x : \emptyset \rightarrow \sigma \rangle$ where *x* is a list of variables, \emptyset a trigger condition and σ a body), etc. Although ConGolog offers programming language-like structures for describing processes, its distinctive feature is that the underlying logic is designed to support reasoning with respect to process specifications and simulations. Figure 15 gives an example of ConGolog specifications for the *checkout* plan graph we have explained previously (see Figure 11).

```

Proc checkoutShoppingCart(shopCart)
  < shopCart : failed(shopCart) → logoutShoppingCart(shopCart) >
  >>
  ( < pressedCancelButton → reinitializeShoppingCart(shopCart) >
    ||
    < timeout > 90 → reinitializeShoppingCart(shopCart) > )
  >>
  < shopCart : ActivatedCheckoutButton ∧ PressedCheckoutButton
    → startCheckOut(shopCart) >
EndProc

```

Figure 15 : ConGolog-like specification for the *checkout* plan from Figure 11

A third language taken into consideration in *Tropos* is KQML (Knowledge Query and Manipulation Language) supported by many agent systems and platforms [Kha99]. KQML is a specification language and protocol for exchanging information and knowledge. It can be used as a communication language [Fin97] to specify agent interactions such as those modeled in Figure 10 (b). The KQML specifications define the syntax and semantics for a collection of messages (or performatives from a speech act point of view) like *achieve*, *ask-if*, *discard*, *register*, *reply*, *stream-about*, *subscribe*, *tell* and keywords that collectively define the language in which agents interact. Such a performative for the *inform* CA from *ShoppingCart* to *Customer* about the final status of the whole *checkout* process (Figure 10 (b)) is defined in Figure 16 considering the case where the process failed.

```

(tell      :language KQML
          :ontology orders
          :content (= (status order9753) nil)
          :force permanent
          :sender shoppingcart7612
          :receiver customer5150)

```

Figure 16: KQML-like specification for the *inform* CA (Figure 10 (b))

10 CONCLUSIONS AND DISCUSSION

We have argued in favour of a software development methodology which is founded on intentional concepts, such as those of actor, goal, (goal, task, resource, softgoal) dependency, etc. Our argument rests on the claim that enterprise software should be organized the same way enterprises are. Moreover, we have argued that current software development techniques lead to inflexible and non-generic software. This is the case because the elimination of goals during late requirements, freezes into the design of a software system a variety of assumptions which may or may not be true in its operational environment. Given the ever-growing demand for generic, component-ized software that can be downloaded and used in a variety of computing platforms around the world, we believe that the use of intentional concepts during late software development phases will become prevalent and should be further researched.

Tropos proposes a modeling framework which views software from five complementary perspectives:

- **Social** -- who are the relevant actors, what do they want? What are their obligations? What are their capabilities?
- **Intentional** -- what are the relevant goals and how do they interrelate? How are they being met, and by whom ask dependencies?
- **Communicational** -- how the actors dialogue and how can they interact with each other?
- **Process-oriented** -- what are the relevant business/computer processes? Who is responsible for what?

- **Object-oriented** -- what are the relevant objects and classes, along with their inter-relationships?

In this paper, we have focused the discussion on the social and intentional perspectives because they are novel. As hinted earlier, we propose to use UML-type modeling techniques for the others.

There already exist some proposals for agent-oriented software development, most notably [Igl98, Jen00, Ode00, Woo00]. Such proposals are mostly extensions to known object-oriented and/or knowledge engineering methodologies. Moreover, all these proposals focus on design -- as opposed to requirements analysis -- for agent-oriented software and are therefore considerably narrower in scope than *Tropos*.

Diagrams are not complete, nor formal as software specifications. To address this deficiency, we propose to offer three levels of software specification. The first is strictly diagrammatic, as discussed in this paper. The second involves formal annotations which complement diagrams. For example, annotations may specify that some obligation takes precedence over another. These could be used as a basis for simple forms of analysis. Finally, we propose to include within *Tropos* a formal specification language for all built-in constructs, to support deeper forms of analysis. Turning to the organization of *Tropos* models, the concepts of *i** will be embedded in a modeling framework which supports generalization, aggregation, classification, materialization and contextualization. Some elements of UML will be adopted as well for modeling the object and process perspectives.

Like other requirements modeling frameworks proposed in the literature, we recognize that diagrams are important for human communication, but are imprecise and offer little support for analysis. Partially formal annotations can help in defining some forms of analysis, and they serve as bridges between informal diagrams and formal specifications. Finally, formal specifications serve as foundation for a formal semantics, as well as a range of analysis techniques, including proofs of correctness, process simulation, goal analysis etc.

ACKNOWLEDGEMENTS

Many colleagues contributed to the ideas that led to this paper. Special thanks to Eric Yu, whose insights helped us focus our research on intentional and social concepts. The Tropos project includes as co-investigators Eric Yu (University of Toronto) and Yves Lespérance (York University); also Alex Borgida (Rutgers University), Matthias Jarke and Gerhard Lakemeyer (Technical University of Aachen). The Canadian component of the project is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the CITO Centre of Excellence, funded by the Province of Ontario. This work was carried out while J. Castro was visiting the Department of Computer Science, University of Toronto (partially supported by the CNPq – Brazil grant 203262/86-7).

REFERENCES

- [Bau99] Bauer, B., *Extending UML for the Specification of Agent Interaction Protocols*, OMG document ad/99-12-03, FIPA submission to the OMG's Analysis and Design Task Force (ADTF) in response to the Request of Information (RFI) entitled "UML2.0 RFI", December 1999.
- [Boo99] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.
- [Bra87] Bratman, M., *Intention, plans, and practical reason*, Harvard University Press, Cambridge, 1987.
- [Cas00] Castro, J., Kolp, M. and Mylopoulos, J., *Developing Agent-Oriented Information Systems for the Enterprise*, *Proceedings of the Second International Conference On Enterprise Information Systems (ICEIS00)*, Stafford, UK, July 2000.

- [Chu00] Chung, L. K., Nixon, B. A., Yu, E. and Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Publishing, 2000.
- [Cob00] Coburn, M., *Jack Intelligent Agents: User Guide version 2.0*, AOS Pty Ltd, 2000.
- [Coh90] Cohen, P. and Levesque, H., "Intention is Choice with Commitment", *Artificial Intelligence*, 32(3), 1990, pp. 213-261.
- [Con00] Conallen, J., *Building Web Applications with UML*, The Addison-Wesley Object Technology Series, Addison-Wesley, 2000.
- [Dar93] Dardenne, A., van Lamsweerde, A. and Fickas, S., "Goal-directed Requirements Acquisition", *Science of Computer Programming*, 20, 1993, pp. 3-50.
- [Dav93] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993.
- [Dem78] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.
- [Fin97] Finin, T., Labrou, Y. and Mayfield, J., "KQML as an Agent Communication Language", Bradshaw, J. M. (ed.), *Software Agents*, MIT Press, 1997, pp. 291-316.
- [Igl98] Iglesias, C., Garrijo, M. and Gonzalez, J., "A Survey of Agent-Oriented Methodologies", *Proceedings of the 5th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL-98)*, Paris, France, July 1998, pp. 317-330.
- [Jen00] Jennings, N. R., "On agent-based software engineering", *Artificial Intelligence*, 117, 2000, pp. 277-296.
- [Kha99] Khalil, C., *Multi-Agent Systems: A Review of Current Technologies*, Department of Computer Science, Loughborough University, IMPACT Research Group, Research Report 99/IMPACT/0182, 1999.
- [Kin96] Kinny, D. and Georgeff, M., "Modelling and Design of Multi-Agent System", *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, Budapest, Hungary, August 1996, pp. 1-20.
- [Kin96a] Kinny, D., Georgeff, M. and Rao, A., "A Methodology and Modelling Technique for Systems of BDI Agents", *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, Eindhoven, The Netherlands, January 1996, pp. 56-71.
- [Les99] Lespérance, Y., Kelley, T. G., Mylopoulos, J. and Yu, E., "Modeling Dynamic Domains with ConGolog", *Proceedings of the 11th Conference on Advanced Information Systems Engineering (CAiSE99)*, Heidelberg, Germany, June 1999, pp. 365-380.
- [My100] Mylopoulos, J. and Castro, J., "Tropos: A Framework for Requirements-Driven Software Development", Brinkkemper, J. and Solvberg, A. (eds.), *Information Systems Engineering: State of the Art and Research Themes*, Springer-Verlag, June 2000.
- [Ode99] Odell, J. and Bock, C., *Suggested UML Extensions for Agents*, OMG document ad/99-12-01, Submitted to the OMG's Analysis and Design Task Force (ADTF) in response to the Request of Information (RFI) entitled "UML 2.0 RFI", December 1999.
- [Ode00] Odell, J., Van Dyke Parunak, H. and Bernhard, B., "Representing Agent Interaction Protocols in UML", *Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, Limerick, Ireland, June 2000.
- [Sho93] Shoham, Y., "Agent-oriented programming", *Artificial Intelligence*, 60, 1993, pp. 51-92.
- [Wir90] Wirfs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Englewood Cliffs, Prentice-Hall, 1990.

- [Woo00] Wooldridge, M., Jennings, N. R. and Kinny D., "The Gaia Methodology for Agent-Oriented Analysis and Design", *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), to appear, 2000.
- [You79] Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.
- [Yu93] Yu, E., "Modeling Organizations for Information Systems Requirements Engineering", *Proceedings of the First IEEE International Symposium on Requirements Engineering*, San Jose, USA, January 1993, pp. 34-41.
- [Yu94] Yu, E. and Mylopoulos, J., "Understanding 'Why' in Software Process Modeling, Analysis and Design", *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 159-168.
- [Yu95] Yu, E., *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.
- [Yu96] Yu, E. and Mylopoulos, J., "Using Goals, Rules, and Methods to Support Reasoning in Business Process Reengineering", *International Journal of Intelligent Systems in Accounting, Finance and Management*, 5(1), January 1996, pp. 1-13.