UML for Agent-Oriented Software Development: The Tropos Proposal*

John Mylopoulos[1], Manuel Kolp[1] and Jaelson Castro[2]

[1]Department of Computer Science, University of Toronto, 10 King's College Road, Toronto M5S 3G4, Canada
{jm, mkolp}@cs.toronto.edu
[2] Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970
jbc@cin.ufpe.br

Abstract. We describe a software development methodology called *Tropos* for agent-oriented software systems. The methodology adopts the *i** modeling framework [29], which offers the notions of *actor*, *goal* and (actor) *dependency*, and uses these as a foundation to model early and late requirements, architectural and detailed design. The paper outlines the methodology, and shows how the concepts of *Tropos* can be accommodated within UML. In addition, we also adopt recent proposals for extensions of UML to support design specifications for agent software. Finally the paper compares *Tropos* to other research on agent-oriented software development.

1 Introduction

The explosive growth of application areas such as electronic commerce, enterprise resource planning and mobile computing has profoundly and irreversibly changed our views on software and Software Engineering. Software must now be based on open architectures that continuously change and evolve to accommodate new components and meet new requirements. Software must also operate on different platforms, without recompilation, and with minimal assumptions about its operating environment and its users. As well, software must be robust and autonomous, capable of serving a naïve user with a minimum of overhead and interference. These new requirements, in turn, call for new concepts, tools and techniques for engineering and managing software.

For these reasons -- and more -- agent-oriented software development is gaining popularity over traditional software development techniques. After all, agent-based architectures (known as multi-agent systems in the Agent research community) *do* provide for an open, evolving architecture which can change at run-time to exploit the services of new agents, or replace under-performing ones. In addition, software agents can, in principle, cope with unforeseen circumstances because they include in their architecture goals, along with a planning capability for meeting them. Finally, agent

* For further detail about the *Tropos* project, see http://www.cs.toronto.edu/km/tropos.

technologies have matured to the point where protocols for communication and negotiation have been standardized [12].

What would it take to adopt a popular software modeling language such as UML [2] and turn it into one that supports *agent*-oriented software development? This paper sketches an agent-oriented software development methodology and proposes extension to UML to accommodate its concepts and features. Our proposal is based on on-going research within the *Tropos* project [3, 23].

Tropos is founded on the premise that in order to build software that operates within a dynamic environment, one needs to analyze and model explicitly that environment in terms of “actors”, their goals and dependencies on other actors.
Accordingly, Tropos supports four phases of software development:

- Early requirements, concerned with understanding the problem by studying an organizational setting; the output of this phase is an organizational model which includes relevant external actors, their respective goals and their inter-dependencies.
- 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, control and other dependencies.
- Detailed design, where behaviour of each architectural component is defined in further detail.

To support modeling and analysis during each of these phases, we adopt the concepts offered by *i** [29], a modeling framework offering concepts such as *actor* (actors can be *agents*, *positions* or *roles*), as well as social dependencies among actors, including *goal*, *softgoal*, *task* and *resource* dependencies. These concepts are used to support modeling during the four phases listed above. This means that both the system’s environment and the system itself are seen as organizations of actors, each having goals to be fulfilled and each relying on other actors to help them with goal fulfillment.

In order to illustrate the *Tropos* software development methodology, we use a small case study for a B2C (business to consumer) e-commerce application. *Media Shop* is a store selling and shipping different kinds of media items such as books, newspapers, magazines, audio CDs, videotapes, and the like. *Media Shop* customers (on-site or remote) can use a periodically updated catalogue describing available media items to specify their order. *Media Shop* is supplied with the latest releases from *Media Producer* 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 world-wide-web using communication facilities provided by *Telecom Co*. It also uses financial services supplied by *Bank Co.*, which specializes on on-line transactions.

The basic objective for the new system is to allow an on-line customer to examine the *Medi@* internet catalogue, and place orders. There are no registration restrictions, or identification procedures for *Medi@* users. 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 are classified into pop, rock, jazz, opera, world, classical music, soundtrack, …) so that customers can browse only (sub)categories of interest. 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 the option of asking *Media Shop* to order it, provided the customer has editor/publisher references (e.g., ISBN, ISSN), and identifies herself (in terms of name and credit card number). For detailed descriptions of the medi@ case study, see [3] and [20].

Section 2 introduces the primitive concepts offered by *i** and illustrates their use for early requirements analysis. Section 3 sketches how the *Tropos* methodology works for later phases of the development process. Section 4 presents fragments of *Tropos* models in UML using existing and extended UML diagrammatic techniques. Section 5 compares our proposal with others in the literature, offers an initial assessment of UML's suitability for modeling agent-oriented software, and outlines directions for further research.

2 Early Requirements with *i**

Early requirements analysis focuses on the intentions of stakeholders. These intentions 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 [8]. In *i** (which stands for "distributed intentionality''), stakeholders are represented as (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 Artificial Intelligence, such as goal, belief, ability, and commitment (e.g., [6]). The framework has been presented in detail in [29] and has been related to different application areas, including requirements engineering [27], business process reengineering [30], and software processes [28].

A strategic dependency model is a graph involving *actors* who have *strategic dependencies* among each other. A dependency describes an "agreement" (called *dependum*) between two actors: the *depender* and the *dependee*. The *depender* is the depending actor, and the *dependee,* the actor who is depended upon. The type of the dependency describes the nature of the agreement. *Goal* dependencies are used to represent delegation of responsibility for fulfilling a goal; *softgoal* dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely (for instance, the appreciation is subjective, or the fulfillment can occur only to a given extent); *task* dependencies are used in situations where the dependee is required to perform a given activity; and *resource* dependencies require the dependee to provide a resource to the depender. As shown in Figure 1, actors are represented as circles; dependums -- goals, softgoals, tasks and resources -- are respectively represented as ovals, clouds, hexagons and rectangles; and dependencies have the form *depender* → *dependum* → *dependee*.

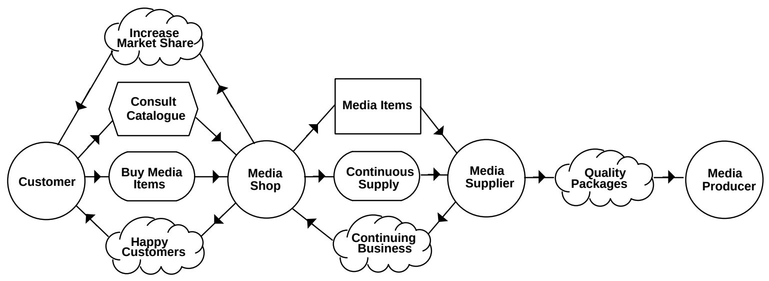

Fig. 1. *i** Model for a Media Shop

These elements are sufficient for producing a first model of an organizational environment. For instance, Figure 1 depicts an *i** model of our *Medi@* example. The main actors are *Customer*, *MediaShop*, *MediaSupplier* and *MediaProducer*. *Customer* depends on *MediaShop* to fulfill her goal: *Buy Media Items*. Conversely, *MediaShop* depends on *Customer* to *increase market* share and make "*customers happy"*. Since the dependum *HappyCustomers* cannot be defined precisely, it is represented as a softgoal.

The *Customer* also depends on *MediaShop* to *consult the catalogue* (task dependency). Furthermore, *MediaShop* depends on *MediaSupplier* to supply media items in a continuous way and get a *Media Item* (resource dependency) . The items are expected to be of good quality because, otherwise, the *Continuing Business* dependency would not be fulfilled. Finally, *MediaProducer* is expected to provide *MediaSupplier* with *Quality Packages.*

We have defined a formal language, called *Formal Tropos* [13], that complements *i** in several directions. First of all, it provides a textual notation for *i** models and allows us to describe dynamic constraints among the different elements of the specification in a first order, linear-time temporal logic. Second, it has a precisely defined semantics that is amenable to formal analysis. Finally, *Formal Tropos* comes with a methodology for the automated analysis and animation of specifications [13], based on model checking techniques [5].

Entity MediaItem
 Attribute constant itemType : ItemType, price : Amount,
 InStock : Boolean

Dependency BuyMediaItems
 Type goal
 Mode achieve
 Depender Customer
 Dependee MediaShop
 Attribute constant item : MediaItem
 Fulfillment
  condition for depender
   $\forall media : MediaItem(self.item.type = media.type \rightarrow item.price <= media.price)$
  [the customer expects to get the best price for the type of item]

Dependency ContinuousSupply
Type goal
Mode maintain
Depender MediaShop
Dependee MediaSupplier
Attribute constant item : MediaItem
Fulfillment
condition for depender
$\exists buy : BuyItem(JustCreated(buy) \rightarrow buy.item.inStock)$
[the media retailer expects to get items in stock as soon as someone is interested in buying them]

Fig. 2. *Formal Tropos* Specifications

As an example, Figure 2 presents the specification in *Formal Tropos* for the *BuyMediaItems* and *ContinuousSupply* goal dependencies. Notice that the *Formal Tropos* specification provides additional information that is not present in the *i** diagram. For instance, the *fulfillment condition* of *BuyMediaItems* states that the customer expects to get the best price for the type of product that she is buying. The condition for *ContinuousSupply* states that the shop expects to have the items in stock as soon as someone is interested in buying them.

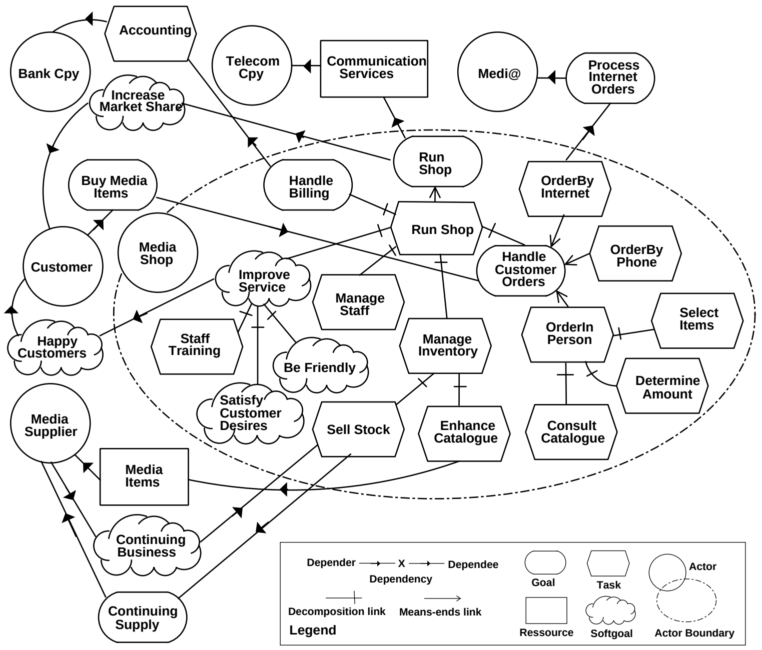

Fig. 3. Means-Ends Analysis for the Softgoal *Increase Market Share*

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 types of nodes -- *goal*, *task*, *resource*, and *softgoal* -- and two types of links -- means-ends links and task decomposition links.   A strategic rationale graph captures the relationship between the goals of each actor and the dependencies through which the actor expects these dependencies to be fulfilled.

Figure 3 focuses on one of the (soft)goal dependency identified for *Media Shop*, namely *Increase Market Share*. To achieve that softgoal, the analysis postulates a goal *Run Shop* that can be fulfilled by means of a task *Run Shop*. 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 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. Sub-goals and subtasks can be specified more precisely through refinement. 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*.

3 Other Phases

3.1 Late Requirements Analysis

Late requirements analysis results in a requirements specification which describes all functional and non-functional requirements for the system-to-be. In *Tropos*, the information 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 who contribute to the fulfillment of stakeholder goals.

For our example, the *Medi@* system is introduced as an actor in the strategic dependency model depicted in Figure 4. With respect to the actors previously identified, *Customer* depends on *Media Shop* to buy media items while *Media Shop* depends on *Customer* to increase market share and remain happy (with *Media Shop* service). *Media Shop* depends on *Medi@* for processing internet orders and on *Bank Cpy* to process business transactions. *Customer*, in turn, depends on *Medi@* to place orders through the internet, to search the database for keywords, or simply to browse the on-line catalogue. With respect to relevant qualities, *Customer* requires that transaction services be secure and usable, while *Media Shop* expects *Medi@* to be easily adaptable. Further dependencies are shown on Figure 4 and explained in [3].

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.

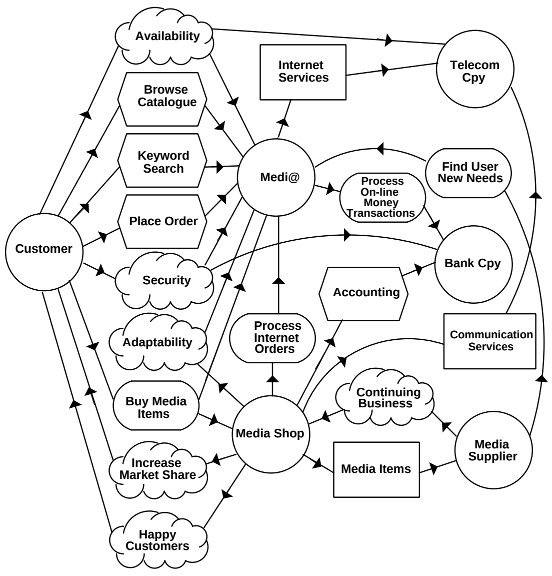

Fig. 4. Strategic Dependency Model for a Media Shop

As late requirements analysis proceeds, *Medi@* 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 responsibility assignment is realized using the same kind of means-ends analysis along with the strategic rationale analysis illustrated in Figure 3. Hence, the analysis in Figure 5 focuses on the system itself, instead of an external stakeholder.

The figure postulates a root task *Internet Shop Managed* providing sufficient support (++) [4] to the softgoal *Increase Market Share*. That 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 *Adaptation*. To manage internet orders, *Internet Order Handled* is achieved through the task *Shopping Cart* which is decomposed into subtasks *Select Item*, *Add Item*, Check *Out*, and *Get Identification Detail*. These are the main process activities required to design an operational on-line shopping cart [7]. The latter (goal) is achieved either through sub-goal *Classic Communication Handled* dealing with phone and fax orders or *Internet Handled* managing secure or standard form orderings. To allow for the ordering of new items not listed in the catalogue, *Select Item* is also further refined into two alternative subtasks, one dedicated to select catalogued items, the other to preorder unavailable products.

To provide sufficient support (++) to the *Adaptable* softgoal, *Adaptability* is refined into four subtasks dealing with catalogue updates, system evolution, interface updates 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 with particular items in mind by using search functions or simply browsing the catalogued products.

In addition, as already pointed, Figure 5 introduces softgoal contributions to model sufficient/partial positive (respectively ++ and +) or negative (respectively - - and -) support to softgoals *Secure*, *Available*, *Adaptable*, *Attract New Customers* and *Increase Market Share*. 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 postulated.

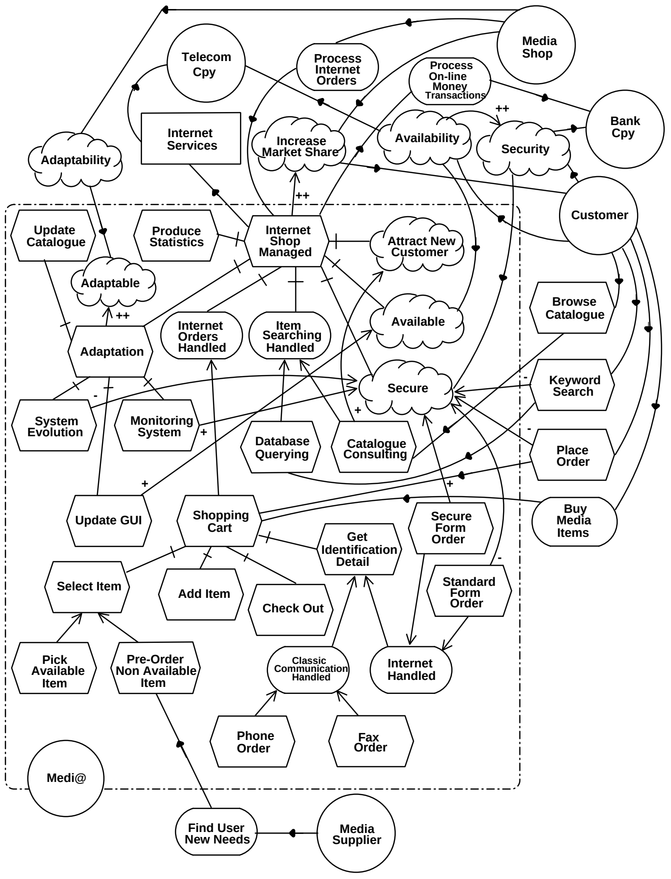

Fig. 5. Strategic Rationale Model for *Medi@*

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 [8], while quality softgoals are either operationalized or "metricized" [9]. For example, 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 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 systems fragile and less reusable.

In our example, we have left three (soft)goals (*Availability, Security, Adaptability*) in the late requirements model. For instance, we have left *Availability* because we propose to allow system agents to automatically decide at run-time which catalogue browser, shopping cart and order processor architecture fit best customer needs or navigator/platform specifications. Moreover, we would like to include different search engines, reflecting different search techniques, and let the system dynamically choose the most appropriate.

3.2 Architectural Design

A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. By now, software architects have developed catalogues of architectural style for e-business applications (e.g., [7]: *Thin Web Client*, *Thick Web Client*, *Web Delivery*, …) Unfortunately, these architectural styles focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organizational architecture styles are not described nor the conceptual high-level perspective of the e-business application. In *Tropos*, we have defined organizational architectural styles [19, 20, 14] for agent, cooperative, dynamic and distributed applications to guide the design of the system architecture. These architectural styles (*pyramid, joint venture, structure in 5, takeover, arm’s length, vertical integration, co-optation, bidding,* …) are based on concepts and design alternatives coming from research on organization management : organization theory, agency theory, strategic alliances, …. For instance, the *joint venture* style involves agreement between two or more principal partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. Through the delegation of authority to a specific *Joint Management* actor that coordinates tasks and manages sharing of knowledge and resources, they pursue joint objectives. Each principal partner can manage and control itself on a local dimension and interact directly with other principal partners to exchange, provide and receive services, data and knowledge. However, the strategic operation and coordination of such a system and its partner actors on a global dimension are only ensured by the *Joint Management* actor.

The first task during architectural design is to select among alternative architectural styles using as criteria the desired qualities identified earlier. The analysis involves refining these qualities, represented as softgoals, to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them, as shown in Figure 6. The styles are represented as operationalized softgoals (saying, roughly, “make the architecture of the new system *pyramid-/joint venture-/co-optation*-based, … ”). Design rationale is represented by claim softgoals drawn as dashed clouds. These can represent contextual information (such as priorities) to be considered and properly reflected into the decision making process. Exclamation marks (! and !!) are used to mark priority softgoals. A check-mark “✔” indicates a fulfilled softgoal, while a cross “✕” labels an unfulfillable one.

Software quality attributes *Security*, *Availability* and *Adaptability* have been left in the late requirements model (See Section 3.1). They will guide the selection process of th appropriate architectural style.

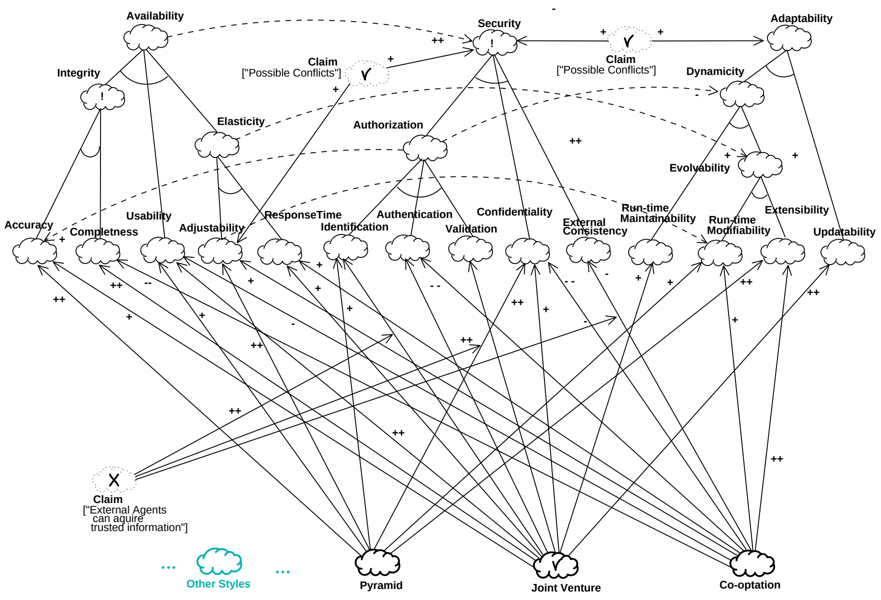

Fig. 6. Selecting the Architecture

In Figure 6, *Adaptability* has been AND-decomposed into *Dynamicity* and *Updatability*. For our e-commerce example, *dynamicity* should deal with the way the system can be designed using generic mechanisms to allow web pages and user interfaces 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. Frameworks like Active Server Pages (ASP), Server Side Includes (SSI) to create dynamic pages make this attribute easier to achieve. *Updatability* should be strategically important for the viability of the application, the stock management and the business itself since *Media Shop* employees have to very regularly bring up to date the catalogue by for inventory consistency. Comparable analyses are carried out in turn for newly identified quality sub-attributes and for the other top-level quality softgoals *Security* and *Availability*.

Eventually, the analysis shown in Figure 6 allows us to choose the joint venture architectural style for our e-commerce example (the operationalized attribute is marked with a "✔"). More details about the selection and non-functional requirements decomposition process can be found in [19, 20]. In addition, more specific attributes have been identified during the decomposition process, such as *Integrity* (*Accuracy*, *Completeness*), *Usability*, *Response Time*, *Maintainability*, *Updatability*, *Confidentiality*, *Authorization* (*Identification*, *Authentication*, *Validation*) and need to be considered in the system architecture.

Figure 7 suggests a possible assignment of system responsibilities, based on the joint venture architectural style for our e-business application. The system is decomposed into three principal partners (*Store Front, Billing Processor* and *Back Store*) controlling themselves on a local dimension and exchanging, providing and receiving services, data and resources with each other.

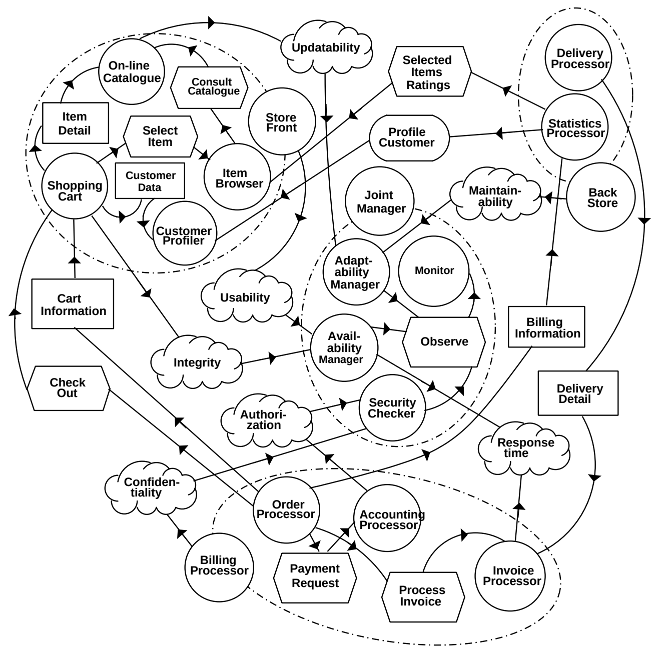

Fig. 7. The E-commerce System in Joint Venture Architecture

Each of them delegates authority to and is controlled and coordinated by the joint management actor (*Joint Manager*) managing the system on a global dimension. *Store Front* interacts primarily with *Customer* 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*. *Joint Manager* manages all of them controlling *security* gaps, *availability* bottlenecks and *adaptability* issues.

To accommodate the responsibilities of *Store Front*, we introduce *Item Browser* to manage catalogue navigation, *Shopping Cart* to select and custom items, *Customer Profiler* to track customer data and produce client profiles, and *On-line Catalogue* to deal with digital library obligations. To cope with the identified software quality attributes (*Security*, *Availability* and *Adaptability*), *Joint Manager* is further refined into four new system sub-actors *Availability Manager*, *Security Checker* and *Adaptability Manager* each of them assuming one of the main softgoals (and their more specific subgoals) and observed by a *Monitor*. Further refinements are shown on Figure 7 and explained in [19, 20].

3.3 Detailed Design

The detailed design phase is intended to introduce additional detail for each architectural component of a system. In our case, this includes actor communication and actor behavior. To support this phase, we propose to adopt agent specifications proposed by FIPA (Foundation for Intelligent Agents) [12] notably agent role and patterns (see [14, 20]) that can be found in agent communication languages like FIPA-ACL [12] or KQML [12].

For instance, the *matchmaker* agent pattern locates a provider corresponding to a consumer request for service, and then hands the consumer a handle to the chosen provider. Contrary to the broker pattern who directly handles all interactions between the consumer and the provider, the negotiation for service and actual service provision are separated into two distinct phases.

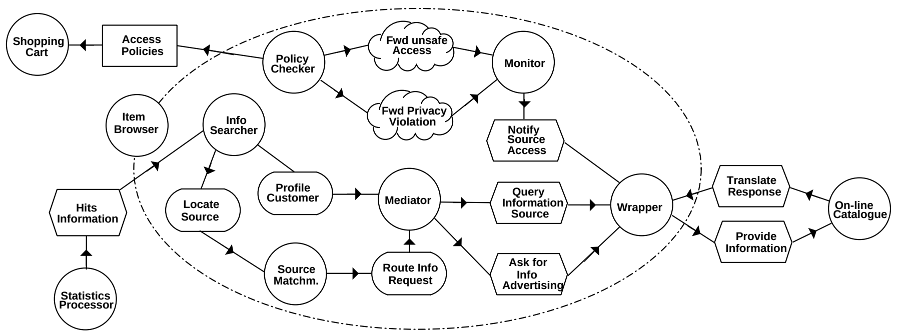

Fig. 8. Detailing Item Browser with Agent Patterns

Figure 8 shows a possible use of the patterns in the e-business system depicted in Figure 7. In particular, it describes how to solve the goal of managing catalogue navigation that the *Store Front* has delegated to the *Item Browser*. The goal is decomposed into different subgoals and solved with a combination of patterns. The broker pattern is applied to the *Info Searcher*, which satisfies requests of searching information by accessing *On-line Catalogue*. The Source *Matchmaker* applies the matchmaker pattern locating the appropriate source for the *Info Searcher*, and the monitor pattern is used to check any possible change in the *On-line Catalogue*. Finally, the mediator pattern is applied to mediate the interaction among the *Info Searcher*, the *Source Matchmaker*, and the *Wrapper*, while the wrapper pattern makes the interaction between the *Item Browser* and the *On-line Catalogue* possible. Of course, other patterns can be applied [20]. For instance, we could use the contract-net pattern to select a wrapper to which delegate the interaction with the *On-line Catalogue*, or the embassy to route the request of a wrapper to the *On-line Catalogue*.

4 *Tropos* Models in UML

We have defined a set of stereotypes, tagged values, and constraints to accommodate *Tropos* concepts within UML. This section briefly describes some of them according to GRL (Goal-oriented Requirement Language) [15]. For an exhaustive and formal definition of the *Tropos* ontology see [15].

Stereotypes

i* actor

Metamodel class	Actor
Description	An actor is an active entity that carries out actions to achieve goals by exercising its know-how. An actor may optionally have a boundary, with intentional elements inside.
Icon	🟢
Constraints	None
Tagged values	Actor_id, external_name, description, goal_model_id

Task

Metamodel class	Use Case
Description	A task specifies a particular way of doing something. Tasks can also be seen as the solutions in the target system, which will satisfice the softgoals (operationalizations). These solutions provide operations, processes, data representations, structuring, constraints and agents in the target system to meet the needs stated in the goals and softgoals.

Icon	⬢
Constraints	None
Tagged values	Task_id, external_name, owner_id, description

Goal

Metamodel class	Class
Description	A goal is a condition or state of affairs in the world that the stakeholders would like to achieve. How the goal is to be achieved is not specified, allowing alternatives to be considered. A goal can be either a business goal or a system goal.
Icon	⬭
Constraints	None
Tagged values	Goal_id, external_name, owner_id, description

i* dependency

Metamodel class	Association
Description	The Dependency statement describes an intentional relationship between two actors, i.e., one actor (<Depender>) depends on another actor (<Dependee>) on something (<Dependum>).
Constraints	Dependencies must have at least one depender and one dependee.
Tagged values	Dependency_id, dependency_name, dependum_type, depender_id, dependee_id

Means-ends

Metamodel class	Association
Description	The Means-ends statement describes how goals are in fact achieved. Each task provided is an alternative means for achieving the goal. Normally, each task would have different types of impacts on softgoals, which would serve as criteria for evaluating and choosing among each task alternative.

Constraints	Only goals are applicable to means-ends links.
Tagged values	Means-ends_id, means_element_id, ends_element_id

Task Decomp.

Metamodel class	Aggregation
Description	The decomposition relationship provides the ability to define what other elements need to be achieved or available in order for a task to perform.
Constraints	Only tasks are decomposable. Sub-components of tasks are goals, tasks, resources, and softgoals.
Tagged values	Decomposition_id, sub-element_id, decomposed_element_id

For instance, Figure 9 depicts the i* model from Figure 1 in UML using the stereotypes we have defined, notably *<<i* actor>>* and *<<i* dependency>>*. Such mapping in UML could also be done in a similar way for strategic rationale (e.g., Figure 3) or goal analysis (e.g., Figure 6) models.

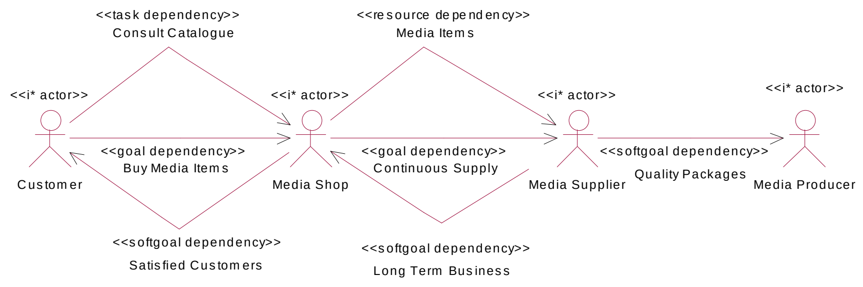

Fig. 9. Representing the *i** Model from Figure 1 in UML with stereotypes

In addition to the introduction of *Tropos* concepts in UML through stereotypes, we also adopt UML extensions proposed by FIPA and the OMG Agent Work group [1, 21, 22]. The rest of the section concentrates on the *Shopping cart* actor and the *check out* dependency. Figure 10 depicts a partial UML class diagram focusing on that actor that will be implemented as an aggregation of several *CartForm*s and *ItemLine*s. Associations *ItemDetail* to *On-line Catalogue,* aggregation of *MediaItem*s, and *CustomerDetail* to *CustomerProfiler*, aggregation of *CustomerProfileCard*s are directly derived from resource dependencies with the same name in Figure 7.

*i** tasks will be implemented as agent plans represented as methods following the label "*Plans*".

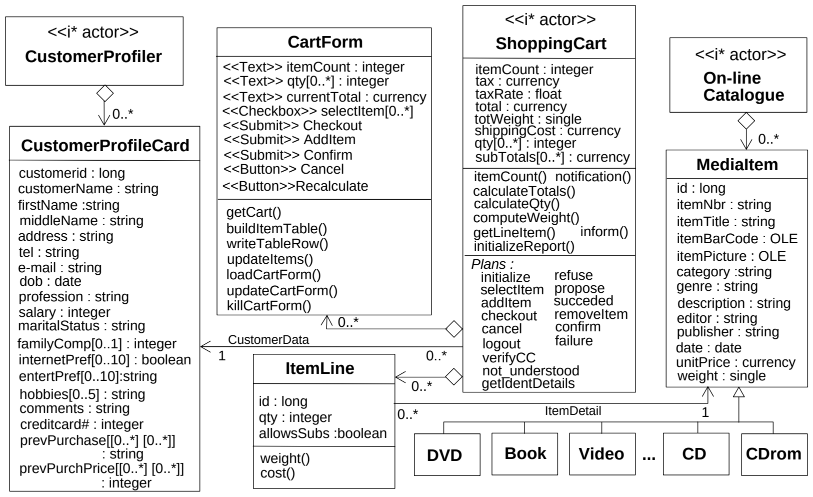

Fig. 10. Partial Class Diagram for *Store Front* Focusing on *Shopping Cart*

To specify the *checkout* task, for instance, we use AUML - the Agent Unified Modeling Language [1], which supports templates and packages to represent *checkout* as an object, but also in terms of sequence and collaborations diagrams.

Figure 11(a) introduces the *checkout* interaction context which is triggered by the *checkout* communication act (CA) and ends with a returned information status. This diagram only provides basic specification for an intra-agent order processing protocol. In particular, 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.

As shown in Figure 11(b), such details can be provided by using *levelling* [22], i.e., by introducing additional interaction and other diagrams. Each additional level can express *inter-actor* or *intra-actor* dialogues. At the lowest level, specification of an actor requires spelling out the detailed processing that takes place within the actor.

Figure 11(b) focuses on the protocol between *Customer* and *Shopping Cart* which consists of a customization of the Contract Net FIPA agent pattern [21]. Such a protocol describes a communication pattern among actors, as well as constraints on the contents of the messages they exchange.

We use plan diagrams [18], based on state charts and activity diagrams, to specify the internal processing (tasks) of atomic actors. The initial transition of the plan diagram 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()*).

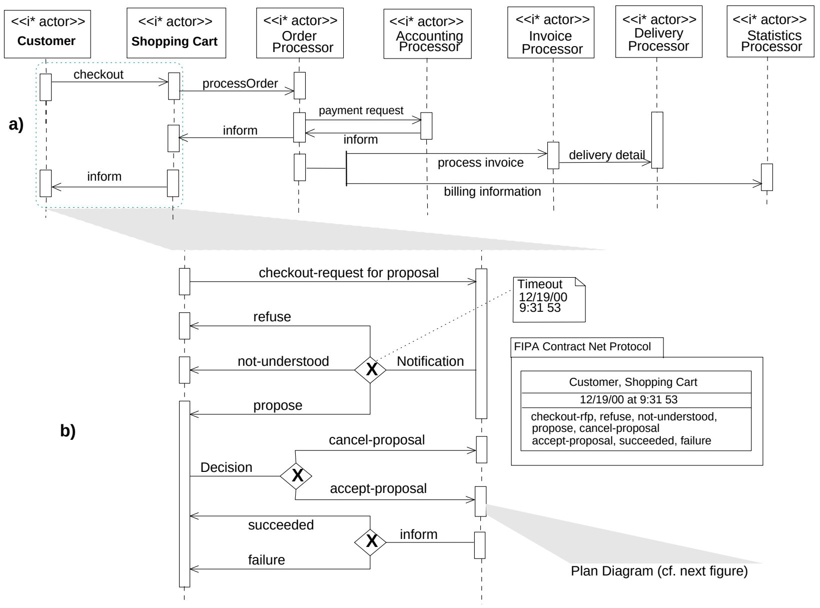
a)
<<i* actor>> Customer
<<i* actor>> Shopping Cart
<<i* actor>> Order Processor
<<i* actor>> Accounting Processor
<<i* actor>> Invoice Processor
<<i* actor>> Delivery Processor
<<i* actor>> Statistics Processor
checkout
processOrder
payment request
inform
inform
process invoice
delivery detail
inform
billing information
b)
checkout-request for proposal
Timeout 12/19/00 9:31 53
refuse
not-understood
Notification
propose
FIPA Contract Net Protocol
Customer, Shopping Cart
12/19/00 at 9:31 53
checkout-rfp, refuse, not-understood, propose, cancel-proposal accept-proposal, succeeded, failure
cancel-proposal
Decision
accept-proposal
succeeded
inform
failure
Plan Diagram (cf. next figure)

Fig. 11. Sequence Diagram to Order Media Items (a), and Agent Interaction Protocol Focusing on a *Checkout* Dialogue (b)

An important feature of plan diagrams is their notion of failure. Failure can occur when an action upon a transition fails, when an explicit transition to a fail state (denoted by a small no entry sign) occurs, or when the activity of an active state terminates in failure and no outgoing transition is enabled.

Figure 12 depicts the plan diagram for *checkout*, triggered by pushing the checkout button. Mandatory fields are first checked. If any 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, the plan continues: each item is iteratively registered, final amounts are calculated, stock records and customer profiles are updated and a report is displayed. When 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 more than 90 seconds (e.g., due to a network bottleneck), the plan fails and the *Shopping Cart* is reinitialized.

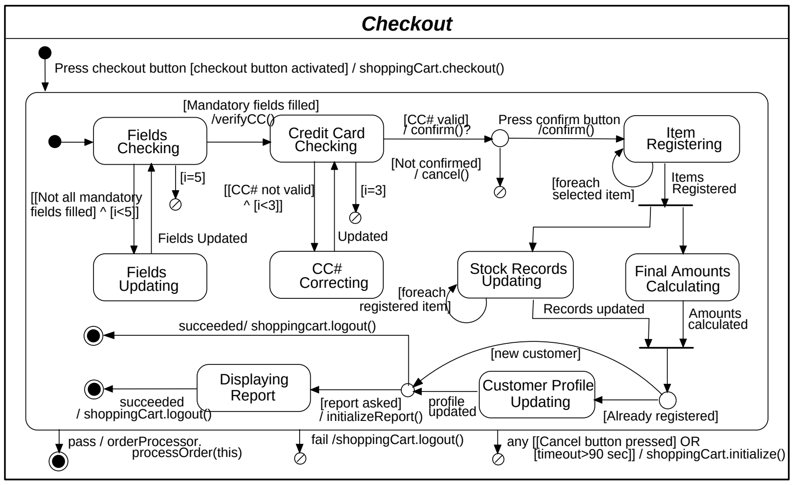
Checkout
Press checkout button [checkout button activated] / shoppingCart.checkout()
[Mandatory fields filled] /verifyCC()
Fields Checking
Credit Card Checking
[CC# valid] / confirm()?
Press confirm button /confirm()
Item Registering
[Not confirmed] / cancel()
[i=5]
[[Not all mandatory fields filled] ^ [i<5]]
[[CC# not valid] ^ [i<3]]
[i=3]
[foreach selected item]
Items Registered
Fields Updated
Updated
Fields Updating
CC# Correcting
Stock Records Updating
Final Amounts Calculating
[foreach registered item]
Records updated
Amounts calculated
succeeded/ shoppingcart.logout()
[new customer]
Displaying Report
Customer Profile Updating
succeeded / shoppingCart.logout()
[report asked] / initializeReport()
profile updated
[Already registered]
pass / orderProcessor. processOrder(this)
fail /shoppingCart.logout()
any [[Cancel button pressed] OR [timeout>90 sec]] / shoppingCart.initialize()

Fig. 12. A Plan Diagram for *Checkout*

5 Conclusion and Discussion

We have proposed a development methodology founded on intentional concepts, and inspired by early requirements modeling. We have also sketched how these concepts can be accommodated within UML, and how they can incorporate recent proposals for UML extensions. We believe that the methodology is particularly appropriate for generic, componentized software systems, such as e-business applications that can be downloaded and used in a variety of operating environments and computing platforms around the world. Preliminary results (e.g., [20, 23]) suggest that the methodology complements well proposals for agent-oriented programming environments.

As a matter of fact, unlike UML and existing UML extensions for agent software development such as AUML [22], the *Tropos* approach is requirement- and goal-oriented, i.e., based and driven by intentional and social primitives. Besides, in *Tropos*, we do not necessarily operationalize or metricize these intentional an social structures early on during the development process, avoiding to freeze solutions to a given requirement in the produced software designs. This kind of approach is especially relevant for developing agent applications since, in addition to be systems requiring flexibility and dynamicity, they are built on mental states like beliefs, intentions, desires or commitments and considered “societies” of software entities.

On the other hand, there already exist some proposals for agent-oriented software development like [10, 16, 17, 18, 25]. 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 -- and are therefore considerably narrower in scope than *Tropos*. Indeed, *Tropos* proposes to adopt the same concepts, inspired by requirements modeling research, for describing requirements *and* system design models in order to narrow the semantic gap between them. The architecture and software design models produced within our framework are intentional in the sense that system components have associated goals that are supposed to be fulfilled. They are also social in the sense that each component has obligations/expectations towards/from other components. Obviously, such models are best suited to cooperative, dynamic and distributed applications like multi-agent systems.

The research reported here is still in progress. Much remains to be done to further refine the proposed methodology and validate its usefulness with real case studies. We are currently working on the development of additional formal analysis techniques for *Tropos* including temporal analysis (using model-checking), goal analysis and social structures analysis, also the development of tools which support different phases of the methodology and the definition of the *Formal Tropos* language.

References

[1] 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.

[2] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[3] Castro, J., Kolp, M. and Mylopoulos, J., "A Requirements-Driven Development Methodology", *Proceedings of the 13th International Conference on Advanced Information Systems Engineering* (CAiSE'01), Interlaken, Switzerland, June 2001.

[4] Chung, L. K., Nixon, B. A., Yu, E. and Mylopoulos, J.*, Non-Functional Requirements in Software Engineering*, Kluwer Publishing, 2000.

[5] Clarke, E., Grumberg, O. and Peled, D., *Model Checking*, MIT Press, 1999.

[6] Cohen, P. and Levesque, H., "Intention is Choice with Commitment", *Artificial Intelligence, 32(3)*, 1990, pp. 213-261.

[7] Conallen, J., *Building Web Applications with UML*, The Addison-Wesley Object Technology Series, Addison-Wesley, 2000.

[8] Dardenne, A., van Lamsweerde, A. and Fickas, S., "Goal–directed Requirements Acquisition", *Science of Computer Programming, 20*, 1993, pp. 3-50.

[9] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993.

[10] DeLoach, S. A. and Wood, M., "Developing Multiagent Systems with agentTool", *Proceedings of the 7th The Seventh International Workshop on Agent Theories, Architectures, and Languages* (ATAL'00), Boston, USA, July, 2000.

[11] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.

[12] *The Foundation for Intelligent Physical Agents,* http://www.fipa.org, 2001.

[13] Fuxman, A., Pistore, M., Mylopoulos, J. and Traverso, P., "Model Checking Early Requirements Specification in Tropos", *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering* (RE'01), Toronto, Canada, August 2001.

[14] Fuxman, A., Giorgini, P., Kolp, M. and Mylopoulos, J., "Information Systems as Social Structures", *Proceedings of the Second International Conference on Formal Ontologies for Information Systems* (FOIS'01), Ogunquit, USA, October 2001.

[15] *Goal Oriented Requirement Language*, http://www.cs.toronto.edu/km/GRL

[16] 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), pp. 317-330, Paris, France, July 1998.

[17] Jennings, N. R., "On agent-based software engineering", *Artificial lntelligence, 117,* 2000, pp. 277-296.

[18] 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), pp. 1-20, Budapest, Hungary, August 1996.

[19] Kolp, M., Castro, J. and Mylopoulos, J., "A Social Organization Perspective on Software Architectures", *Proceedings of the First International Workshop From Software Requirements to Architectures* (STRAW'01), pp. 5-12, Toronto, Canada, May 2001.

[20] Kolp, M., Giorgini, P. and Mylopoulos, J., "A Goal-Based Organizational Perspective on Multi-Agents Architectures", *Proceedings of the 9th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages* (ATAL'01), Seattle, USA, August 2001.

[21] 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.

[22] Odell, J., Van Dyke Parunak, H. and Bauer, B., "Extending UML for Agents", *Proceedings of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence*, pp. 3-17, Austin, USA, July 2000.

[23] Perini, A., Bresciani, P., Giunchiglia, F., Giorgini, P. and Mylopoulos, J., "A Knowledge Level Software Engineering Methodology for Agent Oriented Programming". *Proceedings of the Fifth International Conference on Autonomous Agents* (Agents'01), Montreal, Canada, June 2001.

[24] Wirfs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Englewood Cliffs, Prentice-Hall, 1990.

[25] 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)*, 2000.

[26] Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

[27] Yu, E., "Modeling Organizations for Information Systems Requirements Engineering", *Proceedings of the First IEEE International Symposium on Requirements Engineering* (RE'93), pp. 34-41, San Jose, USA, January 1993.

[28] Yu, E. and Mylopoulos, J., "Understanding 'Why' in Software Process Modeling, Analysis and Design", *Proceedings of the Sixteenth International Conference on Software Engineering* (ICSE'94, , pp. 159-168, Sorrento, Italy, May 1994.

[29] Yu, E., *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.

[30] 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.