A Requirements-Driven Software Development Methodology

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
We propose a software development methodology which is founded on concepts used to model early requirements. Our proposal adopts the *i** modeling framework [21], 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, named *Tropos*, through an example, and sketches a formal language which underlies the methodology and is intended to support formal analysis. The methodology seems to complement well proposals for agent-oriented programming platforms.

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

1 INTRODUCTION

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 [9,19], while object-oriented programming has given rise more recently to object-oriented design and analysis [1,17]. 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. Moreover, such an 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?"

This paper outlines a software development framework, named *Tropos*, 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** [20], 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 for an e-commerce example to model not just early requirements, but also late requirements, as well as architectural and detailed design.

The proposed methodology spans four phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an organizational setting; the output of this phase is an organizational model which includes relevant 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.

The proposed methodology includes techniques for generating an implementation from a *Tropos* detailed design. Using an agent-oriented programming platform for the implementation seems natural, given that the detailed design is defined in terms of (system) actors, goals and inter-dependencies among them. For this paper, we have adopted JACK as programming platform to study the generation of an implementation from a detailed design. JACK is a commercial product based on the BDI (Beliefs-Desires-Intentions) agent architecture. Early previews of the *Tropos* methodology appear in [2, 14].

Section 2 of the paper describes a case study for a B2C (business to consumer) e-commerce application. Section 3 introduces the primitive concepts offered by *i** and illustrates their use with an example. Sections 4, 5, and 6 illustrate how the technique works for late requirements, architectural design and detailed design respectively. Section 7 sketches the implementation of the case study using the JACK agent development environment, while Section 8 discusses the formal language which underlies *Tropos* diagrams. Finally, Section 9 summarizes the contributions of the paper, and relates it to the literature.

2 A CASE STUDY

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 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 Cpy*. It also uses financial services supplied by *Bank Cpy*, which specializes on on-line transactions.

The basic objective for the new system is to allow an on-line customer to examine the items in 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).

3 EARLY REQUIREMENTS WITH *I**

During early requirements analysis, the requirements engineer captures and analyzes 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 [7]. 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., [5]). The framework has been presented in detail in [20] and has been related to different application areas, including requirements engineering [21], software processes [22] and business process reengineering [23].

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 another for something in order that the former may attain some goal. We call the depending actor the *depender* and the actor who is depended upon the *dependee*. The object around which the dependency centers is called the *dependum*. Figure 1 shows the beginning of an *i** model.

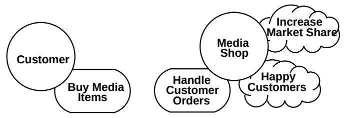


Figure 1: "*Customers* want to buy media items, while the *Media Shop* wants to increase market share, handle orders and keep customers happy"

The two main stakeholders for a B2C application are the consumer and the business actors named respectively in our case *Customer* and *Media Shop*. 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 Customers*, and *Increase Market Share*. Since the last two goals are not well-defined, they are represented as 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 types of nodes -- *goal*, *task*, *resource*, and *softgoal* -- and two types of links -- means-ends links and process 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.

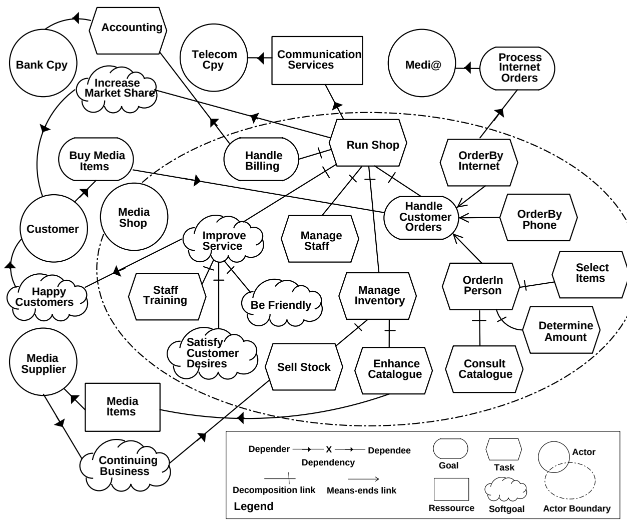


Figure 2: Means-ends analysis for the softgoal *Increase Market Share*

Figure 2 focuses on one of the (soft)goal identified for *Media Shop*, namely *Increase Market Share*. The analysis 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 decomposed into goals *Handle Billing* and *Handle Customer Orders*, tasks *Manage Staff* and *Manage Inventory*, 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 Inventory* would be collectively accomplished by tasks *Sell Stock* and *Enhance Catalogue*.

4 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 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 who 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.

With respect to the actors identified in Figure 2, *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 Supplier* is expected to provide *Media Shop* with media items while depending on the latter for continuing long-term business. He can also use *Medi@* to determine new needs from customers, such as media items not available in the catalogue. As indicated earlier, *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 maintainable (e.g., catalogue enhancing, item database evolution, user interface update, …). The other dependencies have already been described in Figure 2.

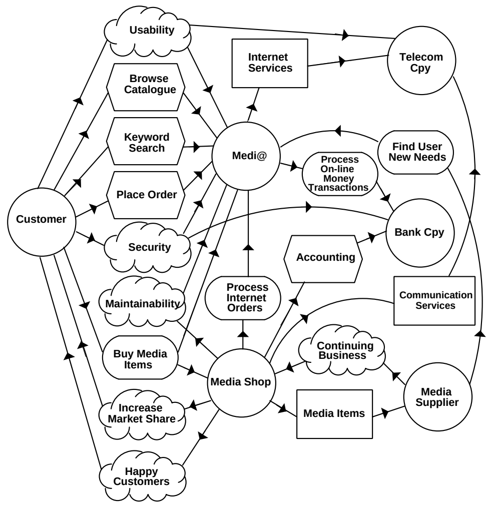


Figure 3: Strategic dependency model for a media shop

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 2. Hence, the analysis in Figure 4 focuses on the system itself, instead of a external stakeholder. The figure postulates a root task *Internet Shop Managed* providing sufficient support (++) [3] 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 *Maintenance*. 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 [6]. 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 selecting catalogued items, the other to preorder unavailable products.

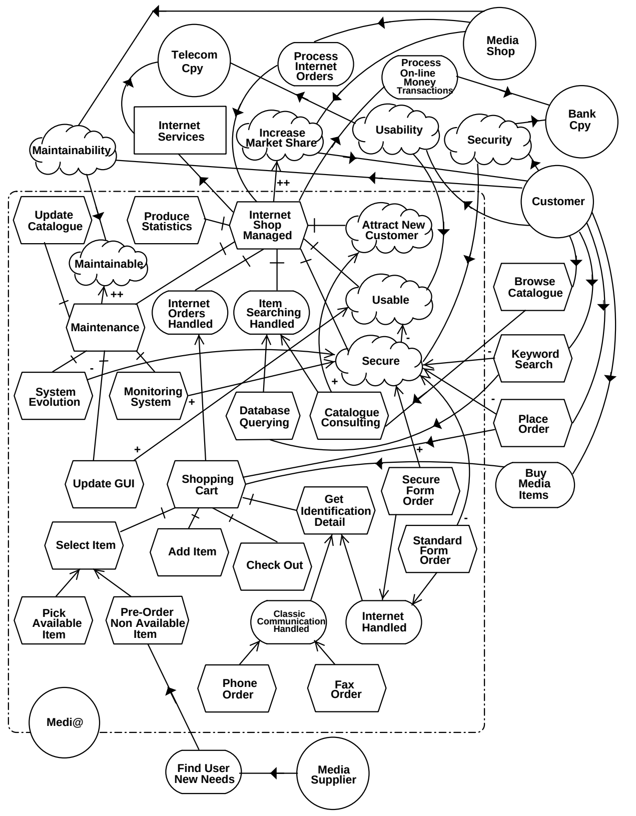


Figure 4: Strategic rationale model for *Medi@*

To provide sufficient support (++) to the *Maintainable* softgoal, *Maintenance* 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 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*. 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.

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

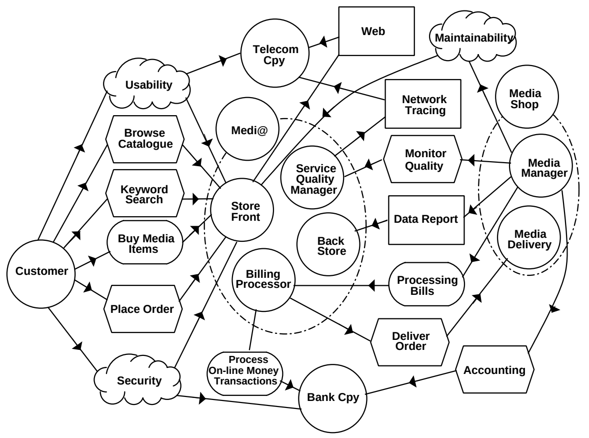


Figure 5: The web system consists of four inside actors, each with external dependencies

Store Front 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*. *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. For instance, *Store Front* communicates to *Billing Processor* relevant customer information required to process bills. 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 selected items. 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, simple browser, frame-based, CGI shopping cart,…)

Finally, *Store Front* 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 *Medi@* may be processing various types of orders differently, and that customers may be selecting the kind of delivery system they would like to use (UPS, FedEx, …).

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 [7], while quality softgoals are either operationalized or "metricized" [8]. 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 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. The second key softgoal in the late requirements specification is *Security*. To fulfil it, we propose to support 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*, meaning that catalogue content, database schema, and architectural model can be dynamically extended to integrate new and future web-related technologies.

5 ARCHITECTURAL DESIGN

A software architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. For our case study, the task is to define (or choose) a web-based 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. This architecture is not intended to preclude the use of distributed objects or Java applets; nor does it imply that the web server and application server cannot be located on the same machine.

By now, software architects have developed catalogues of web architectural style (e.g., [6]). The three most common styles are the *Thin Web Client*, *Thick Web Client* and *Web Delivery*. *Thin Web Client* is most appropriate for applications where the client has minimal computing power, or no control over its configuration. The client requires only a standard forms-capable web browser. *Thick Web Client* extends the *Thin Web Client* style with the use of client-side scripting and custom objects, such as ActiveX controls and Java applets. Finally, *Web Delivery* offers a traditional client/server system with a web-based delivery mechanism. Here the client communicates directly with object servers, bypassing HTTP. This style is appropriate when there is significant control over client and network configuration.

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 *Web Delivery-/Thin Web-/Thick Web*-based") and are evaluated with respect to the alternative non-functional softgoals as shown in Figure 6. 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 while a check-mark "✔" indicates a fulfilled softgoal, while a cross "✕" labels a unfulfillable one.

The *Usability* softgoal has been AND-decomposed into sub-goals *Comprehensibility*, *Portability* and *Sophisticated Interface*. From a customer perspective, it is important for *Medi@* to be intuitive and ergonomic. The look-and-feel of the interface must naturally guides customer actions with minimal computer knowledge. Equally strategic 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. A comparable analysis is carried out for *Security* and *Maintainability*.

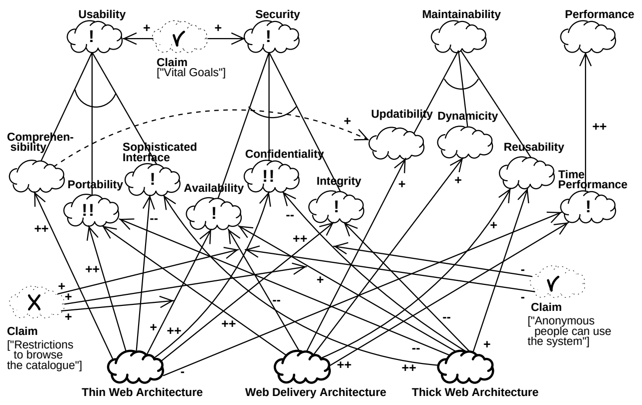


Figure 6: Refining softgoals in architectural design

As shown in Figure 6, each of the three web architectural styles contributes positively or negatively to the qualities of interest. For instance, *Thin Web Client* is useful for applications where only the most basic client configuration can be guaranteed. Hence, this architecture does well with respect to *Portability*. However, it has a limited capacity to support *Sophisticated User Interfaces*. Moreover, this architecture relies on a connectionless protocol such as HTTP, which contributes positively to system availability.

On the other hand, *Thick Web Client* is generally not portable across browser implementations, but can more readily support sophisticated interfaces. As with *Thin Web Client*, all communication between client and server is done with HTTP, 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 client confidentiality. Last but not least, *Web Delivery* 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.

This phase also involves the introduction of new system actors and dependencies, as well as the decomposition of existing actors and dependencies into sub-actors and sub-dependencies which are delegated some of the responsibilities of the key system actors introduced earlier.

Figure 7 focuses on the latter kind of refinement. 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 non-functional requirement decomposition proposed in Figure 6, *Service Quality Manager* is further refined into four new system sub-actors *Usability Manager*, *Security Checker*, *Maintainability Manager* and *Performance Monitor*, each of them assuming one of the top main softgoals explained previously. Further refinements are shown on Figure 7.

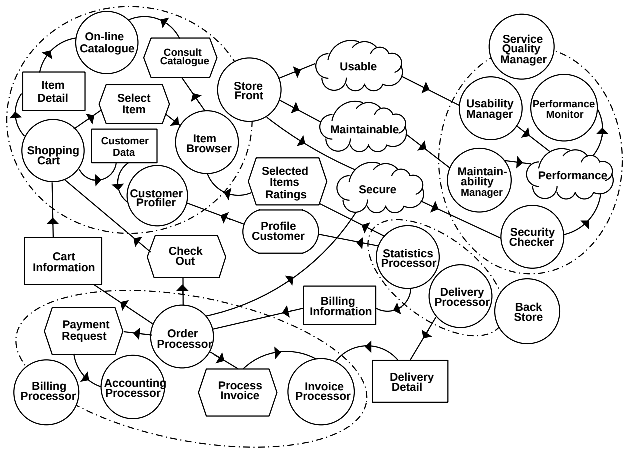


Figure 7: Strategic Dependency Model for *Medi@* actors

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 *Store Front*’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.

6 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 propose to adopt existing agent communication languages, message transportation mechanisms and other concepts and tools. One possibility, for example, is to adopt one of the extensions to UML proposed by the FIPA (Foundation for Intelligent Agents) and the OMG Agent Work group [16]. The rest of the section concentrates on the *Shopping cart* actor and the *check out* dependency. Figure 8 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.

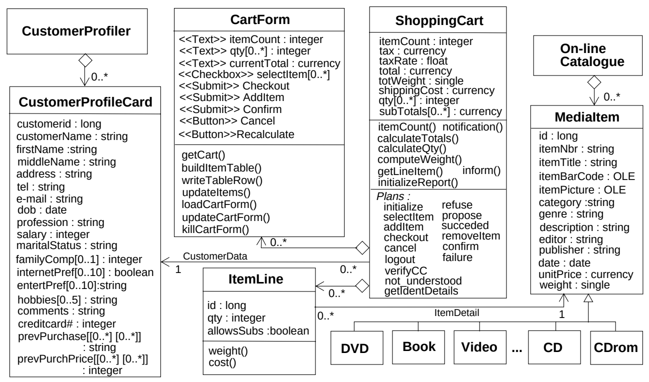


Figure 8: Partial class diagram for *Store Front* focusing on *Shopping Cart*

Our target implementation model is the BDI model, an agent model whose main concepts are *Beliefs*, *Desires* and *Intentions*. As indicated in Figure 11, we propose to implement *i** tasks as BDI intentions (or plans). We represent them as methods (see Figure 8) following the label "*Plans:*".

To specify the *checkout* task, for instance, we use AUML - the Agent Unified Modeling Language [16], which supports templates and packages to represent *checkout* as an object, but also in terms of sequence and collaborations diagrams.

Figure 9(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 by Figure 9(b), such details can be provided by using *levelling* [16], 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.

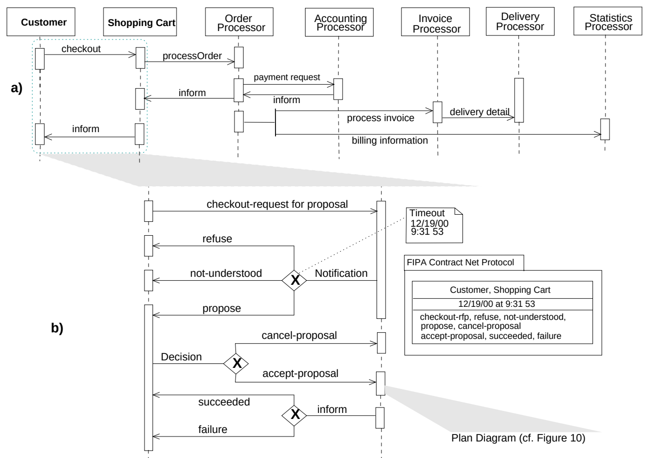


Figure 9: Sequence diagram to order media items (a), and agent interaction protocol focusing on a *checkout* dialogue (b)

Figure 9(b) focuses on the protocol between *Customer* and *Shopping Cart* which consists of a customization of the FIPA Contract Net protocol [16]. Such a protocol describes a communication pattern among actors, as well as constraints on the contents of the messages they exchange.

When a *Custome*r wants to check out, a request-for-proposal message is sent to *Shopping Cart*, which must respond before a given timeout (for network security and integrity reasons). The response may refuse to provide a proposal, submit a proposal, or express miscomprehension. The diamond symbol with an "✕" indicates an "exclusive or" decision. If a proposal is offered, *Customer* has a choice of either accepting or canceling the proposal. The internal processing of *Shopping Cart*'s *checkout* plan is described in Figure 10.

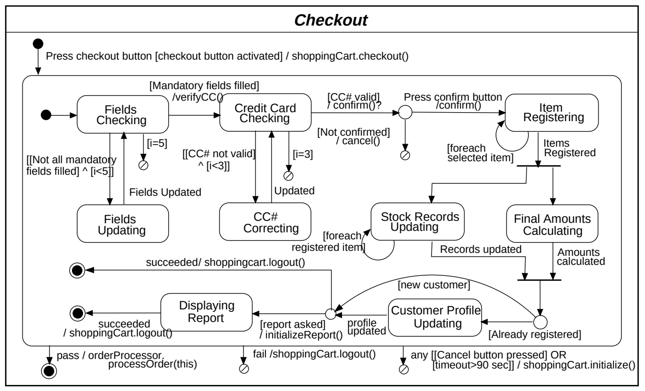

Checkout
Press checkout button [checkout button activated] / shoppingCart.checkout()
[Mandatory fields filled] /verifyCC()
Fields Checking
Credit Card Checking
[CC# valid] / confirm()?
Press confirm button /confirm()
Item Registering
[i=5]
[[Not all mandatory fields filled] ^ [i<5]]
[[CC# not valid] ^ [i<3]]
[i=3]
[Not confirmed] / cancel()
[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()

Figure 10: A plan diagram for *checkout*

At the lowest level, we use plan diagrams [12], to specify the internal processing 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()*).

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 10 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 initializes 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.

7 GENERATING AN IMPLEMENTATION

JACK Intelligent Agents [4] is an agent-oriented development environment designed to provide agent-oriented extensions to Java.

JACK software agents can be considered autonomous software components that have explicit *goals* to achieve, or *events* to cope with (desires). To describe how they should go about achieving these desires, agents are programmed with a set of plans (intentions). Each plan describes how to achieve a goal under different 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 programming of BDI agents, JACK offers five principal language constructs. These are *agents*, *capabilities*, *database relations*, *events*, and *plans*. Figure 11 summarizes the mapping from *i** concepts to JACK constructs and how each concept is related to the others within the same model.

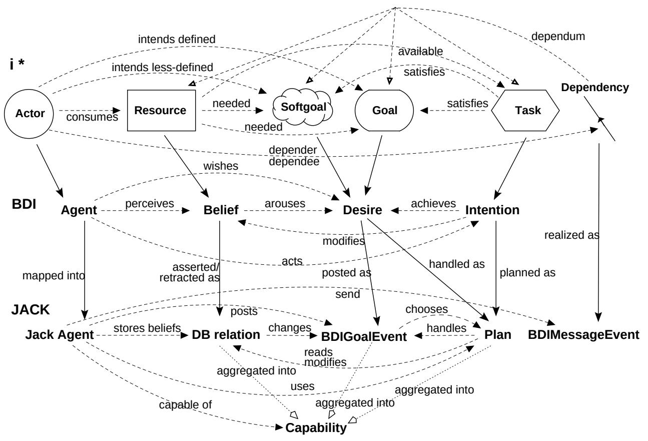
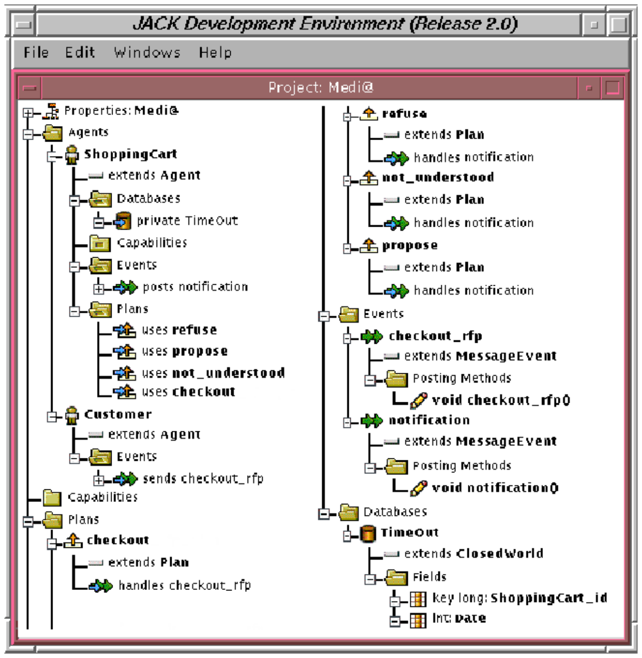


Figure 11: i*/BDI/JACK mapping overview

*I** actors, (informational/data) resources, softgoals, goals and tasks are respectively mapped into BDI agents, beliefs, 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).



```
Properties: Medi@
Agents
  ShoppingCart
    extends Agent
    Databases
      private TimeOut
    Capabilities
    Events
      posts notification
    Plans
      uses refuse
      uses propose
      uses not_understood
      uses checkout
  Customer
    extends Agent
    Events
      sends checkout_rfp
Capabilities
Plans
  checkout
    extends Plan
    handles checkout_rfp
  refuse
    extends Plan
    handles notification
  not_understood
    extends Plan
    handles notification
  propose
    extends Plan
    handles notification
Events
  checkout_rfp
    extends MessageEvent
    Posting Methods
      void checkout_rfp()
  notification
    extends MessageEvent
    Posting Methods
      void notification()
Databases
  TimeOut
    extends ClosedWorld
    Fields
      key long: ShoppingCart_id
      int: Date
```

Figure 12: Partial implementation of Figure 9 in JACK

Figure 12 depicts the JACK layout presenting each of the five JACK constructs as well as the implementation of the first part of the dialogue shown in Figure 10(b). 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*) as detailed in Figure 11. 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

8 A FORMAL LANGUAGE FOR TROPOS

To supplement diagrams with rigorous definitions of the actors, dependencies and relevant entities and relationships, we adopt the language presented in [15]. This language is inspired by KAOS [7] and offers constructs for the specification of the social dependencies supported by i*. It is structured in two layers. The outer layer declares concepts and has an entity-relationship flavour; the inner layer expresses constraints on those concepts in a typed first-order temporal logic with real time constructs [13].

Entity Order
Has orderId: Number, cust: Customer, date: Date,
items: **SetOf** [MediaItem]

Entity MediaItem
Has itemId: Number, itemTitle: String, description: Text, editor: String …

Actor Customer
Has customerId: Number, name: Name, address: Address,
tel: PhoneNumber, …
Capable of MakeOrder, Pay, Browse, …
Goal ∀order:Order ∃buy:BuyMediaItems[order]
(order.cust=self ∧ ◇Fulfil(buy))

Actor MediaShop
Has name: {MediaLive}, address: {"735 Yonge Street"},
phone#: 0461-762-883
Capable of Sell, Ship, SendInvoice, …
Goal ∃ ms:IncreaseMarketShare(Fulfil(ms))

GoalDependency BuyMediaItems
Mode Fulfil
Has order: Order
Defined ItemsReceivedOK(order)
Depender Customer
Dependee MediaShop
Necessary Fulfil(PlaceOrder(order))

SoftGoalDependency IncreaseMarketShare
Mode Maintain
Depender MediaShop
Dependee Customer
Necessary ∀cust:Customer ∃place:PlaceOrder[order]
(order.cust=cust) ∧ ◇Fulfil(place))

Action MakeOrder
Performed By Customer
Refines PlaceOrder
Input cust : Customer, date : Date, items : **SetOf** [MediaItem]
Output order : Order
Post order.cust = cust ∧ order.date = date ∧ order.items ⊆ items

Figure 14: Formal specifications of elements from Figure 3

Figure 14 represents some of the definitions for concepts shown in Figure 3. First, we define the non-intentional entities *Order* and *MediaItem.* Then, we specify the actors in terms of their attributes, goals and the actions they are capable of. For instance, the actor *Customer* can perform the actions *MakeOrder, Pay* and *Browse*. Its goal is that all its orders be successful, in the sense that they should be related to some goal *BuyMediaItems* that is eventually fulfilled.

Goal dependencies are defined in terms of their modality, attributes, involved agents and constraints. The goal *BuyMediaItems* has modality **Fulfil**, which means that it should be achieved at least once. The **Defined** clause gives a formal definition of the goal in terms of the logic predicate *ItemsReceivedOK*. The clause **Necessary** specifies mandatory conditions for the goal to be achieved: in this case, a necessary condition for buying a media item is that the associated order must have been successfully placed. The softgoal *IncreaseMarketShare* has modality **Maintain**, which means that it should never cease to hold. Its (soft) necessary condition is that all the customers eventually place an order.

In addition to the goal model, the language makes it possible to define an operational model that consists of actions that *operationalize (refine)* the goals. Actions are input-output relations over entities, characterized by pre- and post-conditions; action applications define state transitions. As an example, we show the action *MakeOrder* that operationalizes the goal *PlaceOrder* by creating an entity of class *Order*.

9 CONCLUSIONS AND DISCUSSION

We have proposed a software development methodology founded on intentional concepts, and inspired by early requirements modeling. We believe that the methodology is particularly appropriate for generic, componentized software that can be downloaded and used in a variety of operating environments and computing platforms around the world. Preliminary results suggest that the methodology complements well proposals for agent-oriented programming environments.

There already exist some proposals for agent-oriented software development, most notably [10, 11, 16, 18]. 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*.

Of course, 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 formal analysis techniques for *Tropos*, also the development of tools which support different phases of the methodology.

ACKNOWLEDGEMENTS

The Tropos project has been partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and Communications and Information Technology

Ontario (CITO), a centre of excellence, funded by the province of Ontario.

We are grateful to Ariel Fuxman and Eric Yu (University of Toronto) for helpful suggestions and feedback to this research.

This work was carried out while Jaelson Castro was visiting the Department of Computer Science, University of Toronto (partially supported by the CNPq – Brazil grant 203262/86-7).

REFERENCES

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

[2] 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.

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

[4] Coburn, M., *Jack Intelligent Agents: User Guide version 2.0*, AOS Pty Ltd, 2000.

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

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

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

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

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

[10] 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.

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

[12] 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.

[13] Koymans, R, Specifying message passing and time-critical systems with temporal logic. In Springer-Verlag LNCS 651. Springer-Verlag, Berlin, 1993.

[14] 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, pp. 261-273.

[15] Mylopoulos, J., Fuxman, A. and Giorgini, P. From Entities and Relationships to Social Actors and Dependencies. To appear in *Proceedings of the 19th International Conference on Conceptual Modeling* (ER2000), Salt Lake City, USA, October 2000.

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.