The Tropos Methodology: An Overview [†]

Paolo Giorgini[§]
Department of Information and Communication Technology
University of Trento, Italy - paolo.giorgini@dit.unitn.it

Manuel Kolp
Information Systems Research Unit - School of Management
University of Louvain, Belgium - kolp@isys.ucl.ac.be

John Mylopoulos
Department of Computer Science
University of Toronto, Canada - jm@cs.toronto.edu

Marco Pistore
Department of Information and Communication Technology
University of Trento, Italy - marco.pistore@dit.unitn.it

Abstract. The objective of this paper is to give an overview of Tropos methodology. Tropos is based on two key ideas. First, the notion of agent and related mentalistic notions, such as goals and plans, are used in all phases of software development, from early analysis down to the actual implementation. Second, Tropos covers the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software-to-be will eventually operate. We illustrate the phases of the methodology, the Formal Tropos language, and the social and intentional models that are used to support software development.

Keywords: Multi-agent systems, software development, requirements engineering, goal analysis, model checking, architectural styles.

1. Introduction

The explosive growth of application areas such as electronic commerce, enterprise resource planning, and peer-to-peer computing has deeply 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

[†] This research has been partly supported by the Italian MIUR-FIRB Project, RBNE0195K5 - ASTRO and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[§] Contact author. Address: Dipartimento di Informatica - Via Sommarive 14 - 38050 Povo (Trento) - Italy. Phone: ++39 0461 882052.

and autonomous, capable of serving end users 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, including structured and object-oriented ones (see for instance [14]). After all, agent-based architectures *do* provide for an open, evolving architecture that 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 their architecture includes goals along with a planning capability for meeting them.

We are currently working on an agent-oriented software development methodology called Tropos [2]. In a nutshell, Tropos is based on two key features. First, the notion of agent and related mentalistic notions are used in all software development phases, from the early requirements analysis down to the actual implementation. Second, the methodology emphasizes early requirements analysis, the phase that precedes the prescriptive requirements specification. In this respect, Tropos is quite different from other agent- and object-oriented software development methodologies.

Paying attention to the activities that precede the specification of prescriptive requirements for the system-to-be [6, 23] means that developers can capture and analyze the goals of stateholders. These goals play a crucial role in defining the requirements for the new system. Put another way, prescriptive requirements capture the *what* and the *how* for the system-to-be. Early requirements, on the other hand, capture the reasons *why* a software system is developed. This new perspective, in turn, supports a more refined analysis of system dependencies and a more uniform treatment of functional and non-functional requirements.

Tropos adopts Eric Yu's *i** model which offers actors (agents, roles, or positions), goals, and actor dependencies as primitive concepts for modeling an application during early requirements analysis. Tropos is intended to support four phases of software development: *early requirements* analysis, concerned with the understanding of a problem by studying its organizational setting; *late requirements* analysis, 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; and *detailed design*, where the behavior of each component is defined in further detail.

The objective of this chapter is to present the Tropos methodology. Section 2 offers an overview of the methodology, while Section 3 presents the Tropos formal language, designed to support the methodology. Section 4 describes the social patterns used during the development process, while Section 5 presents its goal model. Finally, Section 6 summarizes the contributions of the proposed methodology and points to directions for further work.

2. Overview

In this section we present briefly the four phases supported by Tropos, using the *Media Shop* case study. *Media Shop* is a store selling and shipping media items such as books, magazines, audio CDs, videotapes, and the like. *Media Shop* customers (on-site or remote) can use a catalogue describing available items to fill their orders. *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 *Medi@*, a B2C internet site. Through it, a customer can put in orders to *Media Shop* through the internet. She can also search the on-line store by either browsing the catalogue, or by querying the database through keywords or full-text search. The system uses communication facilities provided by *Telecom Cpy* and on-line financial services supplied by *Bank Cpy*.

Early requirements analysis focuses on the intentions of stakeholders. Intentions are modeled as goals. Through some form of goal-oriented analysis, these initial goals eventually lead to the functional and non-functional requirements of the system-to-be [6]. In *i** [23], 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.

A strategic dependency model is a graph involving *actors* who have *strategic dependencies* among each other. A dependency describes an “agreement” (called *dependum*) between a depending actor (*depender*) and an actor who is depended upon (*dependee*). 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

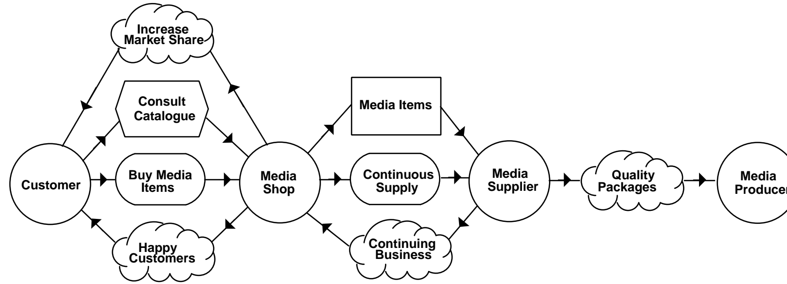

Figure 1. i^* Model for a Media Shop.

fulfillment cannot be defined precisely (for instance, the degree of fulfillment is subjective); *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*.

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*, *Media Shop*, *Media Supplier* and *Media Producer*. *Customer* depends on *Media Shop* to fulfill her goal: *Buy Media Items*. Conversely, *Media Shop* depends on *Customer* to *increase market share* and make "*customers happy*". Since the dependum *Happy Customers* cannot be defined precisely, it is represented as a softgoal. The *Customer* also depends on *Media Shop* to *consult the catalogue* (task dependency). Furthermore, *Media Shop* depends on *Media Supplier* to supply media items in a continuous way and get a *Media Item* (resource dependency). The items are expected to be of good quality, otherwise the *Continuing Business* dependency might not be fulfilled. Finally, *Media Producer* is expected to provide *Media Supplier* with *Quality Packages*.

Late requirements analysis results in a requirements specification which describes all functional and non-functional requirements for the system-to-be. In Tropos, the 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.

As late requirements analysis proceeds, the system (*Medi@*) is given additional responsibilities, and ends up as the dependee of several dependencies. A strategic rationale model determines through a means-

ends analysis how the system goals (including softgoals) identified during early requirements can actually be fulfilled exploiting 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 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.

The analysis in Figure 2 focuses on the system itself and postulates a root task *Internet Shop Managed* providing sufficient support (++) [3] to the softgoal *Increase Market Share*. That task is firstly refined (through decomposition links) into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Security*, *Adaptability* and *Availability*, and task *Produce Statistics*. To manage internet orders, *Internet Order Handled* needs to be achieved (means-ends link) through the task *Shopping Cart*. In turn, this task is decomposed into subtasks *Select Item*, *Add Item*, *Check Out,* and a subgoal *Get Identification Detail*. These are the main process activities required to design an operational on-line shopping cart. The latter goal is achieved either through secure or standard form orderings.

In addition, Figure 2 introduces softgoal contributions to model sufficient/partial positive (respectively ++ and +) or negative (respectively −− and −) support to softgoals *Security*, *Availability*, *Adaptability*, *Attract New Customers* and *Increase Market Share*. The result of such a means-ends analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been postulated.

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, while quality softgoals are either operationalized or “metricized” [6]. In our example, we have left four (soft)goals (*Availability, Security, Adaptability* and *Increase Market Share*) for architectural design. The operationalization of these non-functional requirements will depend on the type of architecture chosen during design.

Architectural design. A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together [22]. In Tropos, we have defined organizational architectural styles [15] for cooperative, dynamic and distributed applications – such as multi-agent systems – to guide the design of the system architecture. These organizational architectural styles are based on concepts and design alternatives coming from

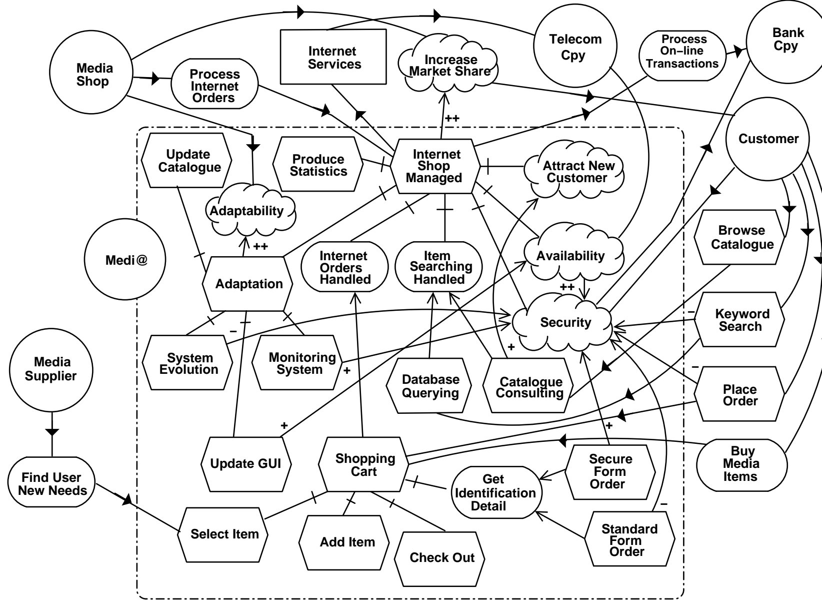

Figure 2. Strategic Rationale Model for *Medi@*.

research in organization management. As such, they help match a multi-agent system (hereafter MAS) architecture to the organizational context within which the system will operate. We present more details on these organizational styles and the Tropos architectural design phase in Section 4.

Detailed design introduces additional detail for each architectural component of a system. In particular, this phase determines how the goals assigned to each actor are fulfilled by agents in terms of design patterns. Design patterns (e.g., [11]) have attracted much attention. Unfortunately, the literature focuses on object-oriented patterns, rather than the intentional and social ones that are relevant here. Within Tropos, social patterns [7] are used to find a solution to a specific goal defined at the architectural level through the identification of organizational styles and relevant quality attributes. More details about social patterns are presented in Section 4.

Detail design in Tropos also includes the specification of agent communication and agent behavior. To support this task, we propose to adopt existing agent communication languages, such as FIPA-ACL [16], and extensions to UML, such as the Agent Unified Modeling Language (AUML).

3. Formal Tropos

The Tropos framework supports the application of formal analysis techniques for the verification of requirements specifications. The analysis is based on *Formal Tropos* (hereafter FT), a specification language that offers all the standard mentalistic notions of Tropos and supplements them with a rich temporal specification language inspired by KAOS [17]. FT allows for the description of the dynamic aspects of Tropos models. More precisely, in FT we focus not only on the intentional elements themselves, but also on the circumstances in which they arise, and on the conditions that lead to their fulfillment. In this way, the dynamic aspects of a requirements specification are introduced at the strategic level, without requiring an operationalization of the specification. With an FT specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfill the goals of the actors? Do the dependencies represent a valid synchronization between actors?

In this section we give a short description of the key aspects of the FT language. A full definition can be found in [8, 10].

An FT specification describes the relevant elements (actors, goals, dependencies...) of a domain and the relationships among them. The description of each elements is structured in two layers. The outer layer is similar to a class declaration. It associates to the element a set of attributes that define its structure. The inner layer expresses constraints on the lifetime of the objects, given in a typed first-order linear-time temporal logic.

Figure 3 is an excerpt of the outer layer of the FT specification of the *Medi@* example. It focuses on the management of the on-line shopping cart. Actors, intentional elements, and dependencies of the Strategic Rational Model are mapped into corresponding "classes" in the outer layer of FT. Moreover, "entities" (e.g., Cart and Item) are added to represent the relevant non-intentional elements of the domain. Several instances of a class may exist during the evolution of the system. For example, different PlaceOrder instances may exist for different customers, and several AddItem tasks can be done during the management of a ShoppingCart.

Each class has an associated list of attributes. Most of the attributes in FT are references to other classes and are used to define the relationships among the different instances of these classes. For example, task ShoppingCart refers to the specific Cart that is being managed (attribute cart) and to the PlaceOrder dependency that triggered the management of the ShoppingCart (attribute po). Moreover, each Cart refers to the set of items that have been added to it. **Constant** attributes (i.e.,

```
Entity Item
Entity Cart
  Attribute items : set of Item
Actor Customer
Actor Medi@
Goal Dependency PlaceOrder
  Depender Customer
  Dependee Medi@
  Mode achieve
Task ShoppingCart
  Actor Media@
  Mode achieve
  Attribute constant cart : Cart
            constant po : PlaceOrder
Task AddItem
  Actor Student
  Mode achieve
  Attribute constant sc : ShoppingCart
            constant item : Item
Goal GetIdentificationDetail
  Actor Medi@
  Mode achieve
  Attribute constant sc : ShoppingCart
            customer : Customer
SoftGoal Security
  Actor Medi@
  Mode maintain
```

Figure 3. Excerpt of FT Class Declaration.

attributes whose values do not change over time) define static relations among the class instances of a model. For instance, the cart associated to a given instance of ShoppingCart does not change. The set of items associated to the cart, on the other hand, can change over time. Special attribute **actor** associates a goal or task to the corresponding actor. Similarly, **depender** and **dependee** attributes define the two actors involved in a dependency.

Intentional elements have a **Mode** attribute that defines the modality of the fulfillment of the goal or task. For instance, the mode of task ShoppingCart is **achieve**, which means that the Medi@ actor wants to reach a state where the management of the cart has been fulfilled and the corresponding order has been placed. Softgoal Security, instead, has a **maintain** mode, since the security of the system has to be continuously maintained.

```
Task AddItem
  Actor Medi@
  Mode achieve
  Attribute constant sc : ShoppingCart
            constant item : Item
  Invariant sc.actor = actor
  Invariant ∀ ai : AddItem ((ai.item = item) → (ai = self))
  Creation condition !Fulfilled (sc)
  Fulfillment definition item in sc.cart.items

SoftGoal Security
  Actor Medi@
  Mode maintain
  Fulfillment condition ∀ gid : GetIdentificationDetail
    (gid.actor = actor ∧ Fulfilled (gid) →
      gid.customer = gid.sc.po.depender)
```

Figure 4. Example of FT Constraints.

Figure 4 contains some examples of constraints on the lifetime of class instances that define the inner layer of an FT specification. **Invariant** constraints define conditions that should be true throughout the lifetime of class instances. Typically, invariants define relations on the possible values of attributes, or cardinality constraints on the instances of a given class. For instance, the first invariant of Figure 4 binds an AddItem task with its associated ShoppingCart task, while the second invariant imposes a cardinality constraint on the AddItem tasks for a given Item.

Two critical moments in the lifecycle of intentional elements and dependencies are the instants of their creation and fulfillment. Creation and fulfillment constraints can be used to impose conditions for these two moments in the life of an intentional element. The creation of a goal or task instance means that the owner or depender expects or desires the achievement of the goal/task. **Creation** constraints should be satisfied whenever a new instance is created, while **fulfillment** constraints should hold whenever a goal or softgoal is satisfied, a task is performed, a resource is made available, or a dependum is delivered. Creation and fulfillment constraints are further distinguished as sufficient conditions (keyword **trigger**), necessary conditions (keyword **condition**), and necessary and sufficient conditions (keyword **definition**).

A first usage of creation and fulfillment constraints is to relate subordinate goals and tasks with their parent intentional elements. For instance, Figure 4 shows that a creation condition for an instance of task AddItem is that the parent task ShoppingCart is not yet fulfilled:

it is not possible to add further items to a cart, once an order has been placed and task ShoppingCart has been fulfilled. Together with the fulfillment conditions of task ShoppingCart, this creation condition elaborates the decomposition relation between the two tasks shown in Figure 2.

The fulfillment condition of softgoal Security requires that, whenever a GetIdentificationDetail goal has been fulfilled, the identified customer (gid.customer) coincides with the customer that is interacting with the system for placing the order (gid.sc.po.**depender**), that is, in a secure system we do not allow for invalid identifications.

Once a FT specification has been defined, it can be formally verified in order to identify errors, ambiguities, and under-specifications. The verification phase usually generates feedback on errors in the FT specification and hints on how to fix them. In order to support the verification process, we have developed a prototype tool, called the T-Tool [9], that is based on finite-state model checking [5, 4]. On the basis of an FT specification, the T-Tool builds a finite model that represents all possible behaviors of the domain that satisfy the constraints of the specification. The T-Tool then verifies whether this model exhibits the desired behaviors.

The T-Tool provides several verification functionalities. *Animation* of the specification consists of an interactive generation of a valid scenario, namely, of a scenario that satisfies all the temporal constraints of the FT specification. Animation allows for an immediate feedback on the effects of constraints and for an early identification of trivial bugs and missing requirements. *Consistency checks* verify that the FT specification is not self-contradictory. Inconsistent specifications occur quite often due to complex interactions among constraints in the specification, and they are very difficult to detect without the support of automated analysis tools. During the consistency checks, the T-Tool verifies that there is some valid scenario that respects all the constraints of the FT specification, that all the goals and dependencies are fulfillable in some scenarios, and other similar properties. *Possibility checks* verify whether we are over-constraining the specification, that is, whether we have ruled out scenarios expected by the stakeholders. These expected scenarios are described in the FT specification using **possibility** properties. For instance, a scenario that we do not want to rule out is the possibility of interrupting the placement of an order also if we have already added some items to the cart. This property can be expressed by the following FT **possibility**. It requires that, even if items have been added to the cart, it is possible to never fulfill a

ShoppingCart task (with "**globally** (c)" we specify that condition c is true through all future history of the model):

$$\textbf{Possibility}\ \exists\ \mathsf{sc: ShoppingCart}\ (\mathsf{sc.cart} \neq \textbf{empty} \land \textbf{globally}\ (\neg\ \textbf{Fulfilled}\ (\mathsf{sc})))$$

Assertion properties verify whether the requirements are under-specified and allowing for invalid scenarios. Also in this case, **assertion** declarations in the FT specification are used to express conditions on the valid scenarios. For instance, a requirement that one wants to be true is that the system is secure, that is, that softgoal Security is fulfilled:

$$\textbf{Assertion}\ \forall\ \mathsf{sec: Security}\ (\textbf{Fulfilled}\ (\mathsf{sec}))$$

Since the fulfillment of the security goal depends on the success of goal GetIdentificationDetail, the definition of the fulfillment conditions of this goal need special care. If these conditions do allow for incorrect identifications, the previous assertion is violated and an error is reported during the verification phase.

4. Socially-Based MAS Architectures

System architectural design has been the focus of considerable research during the last fifteen years. This has produced well-established architectural styles and frameworks for evaluating their effectiveness with respect to particular software qualities. Examples of styles are pipes-and-filters, event-based, layered, control loops and the like [22]. In Tropos, we are interested in developing a suitable set of architectural styles for multi-agent software systems. Since the fundamental concepts of a Multi-Agent System (MAS) are intentional and social, rather than implementation-oriented, we turn to theories which study social structures that result from a design process, namely *Organization Theory* and *Strategic Alliances*. Organization Theory (e.g., [21]) describes the structure and design of an organization; *Strategic Alliances* (e.g., [19]) models the strategic collaborations of independent organizational stakeholders who have agreed to pursue a set of business goals.

Organization Theory describes how practical organizations are actually structured, offers suggestions on how new ones can be constructed, and how old ones can change to improve effectiveness. To this end, schools of organization theory have proposed models such as the *structure-in-5*, the *pyramid style*, the *chain of values*, the *matrix*, the *bidding style* to try to find and formalize recurring organizational structures and behaviors.

For instance the **structure-in-5**, as proposed by Minztberg [18], specifies that an organization is an aggregate of five sub-structures. At the base level sits the *Operational Core* which carries out the basic tasks and procedures directly linked to the production of products and services (acquisition of inputs, transformation of inputs into outputs, distribution of outputs). At the top lies the *Strategic Apex* which makes executive decisions ensuring that the organization fulfills its mission in an effective way and defines the overall strategy of the organization in its environment. The *Middle Line* establishes a hierarchy of authority between the Strategic Apex and the Operational Core. It consists of managers responsible for supervising and coordinating the activities of the Operational Core. The *Technostructure* and the *Support* are separated from the main line of authority and influence the operating core only indirectly. The Technostructure serves the organization by making the work of others more effective, typically by standardizing work processes, outputs, and skills. It is also in charge of applying analytical procedures to adapt the organization to its operational environment. The Support provides specialized services, at various levels of the hierarchy, outside the basic operating work flow (e.g., legal counsel, R&D, payroll, cafeteria).

Figure 5 suggests a possible assignment of system responsibilities for our *Medi@* case study following the structure-in-5 organizational style. It is decomposed into five principal components *Store Front*, *Coordinator*, *Billing Processor*, *Back Store* and *Decision Maker*. *Store Front* serves as the *Operational Core.* It interacts primarily with Customer and provides her with a usable front-end web application for consulting and shopping media items. *Back Store* constitutes the *Support* component. It manages the product database and communicates to the *Store Front* information on products selected by the user. It *stores* and *backs up* all web information from the *Store Front* about customers, products, sales, orders and bills to produce *statistical information* to the *Coordinator*. It provides the *Decision Maker* with *strategic information* (analyses, historical charts and sales reports).

The *Billing Processor* is in charge of handling orders and bills for the *Coordinator* and implementing the corresponding procedures for the *Store Front.* It also ensures the secure management of financial transactions for the *Decision Maker.* As the *Middle Line*, the *Coordinator* assumes the central position of the architecture. It ensures the coordination of *e-shopping* services provided by the *Operational Core* including the management of conflicts between itself, the *Billing Processor*, the *Back Store* and the *Store Front.* To this end, it also handles and implements strategies to manage and prevent *security* gaps and *adaptability* issues. The *Decision Maker* assumes the *Strategic Apex*

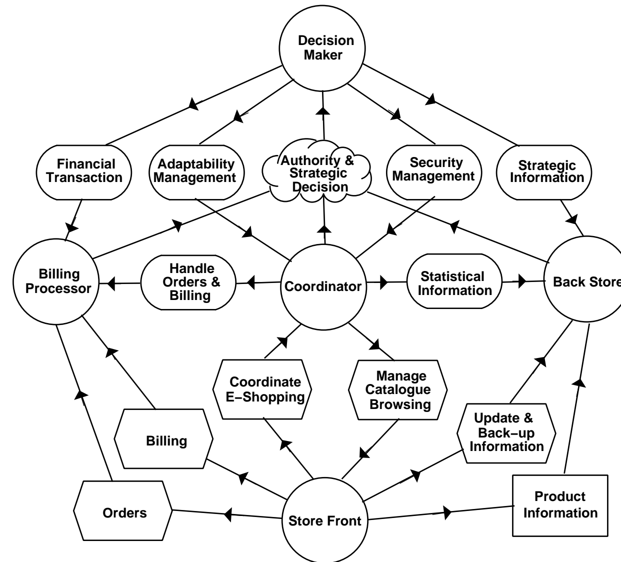

Figure 5. The Medi@ Organizational Architecture in Structure-in-5.

role. To this end, it defines the *Strategic Behavior* of the architecture ensuring that objectives and responsibilities delegated to the *Billing Processor*, *Coordinator* and *Back Store* are consistent with that global functionality.

Strategic Alliances link specific facets of two or more organizations. At its core, this structure is a trading partnership that enhances the effectiveness of competitive strategies of participant organizations by providing for the mutually beneficial trade of technologies, skills, or products derived from them.

For instance, the **joint venture style** involves agreement between two or more intra-industry partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. A specific joint management actor coordinates tasks and manages the sharing of resources between partner actors. Each partner can manage and control itself on a local dimension and interact directly with other partners to exchange resources, such as data and knowledge. However, the strategic operation and coordination of such an organization, and its actors on a global dimension, are only ensured by the joint management actor in which the original actors possess equity participations.

Other styles are the arm's-length style, the hierarchical contracting style or the co-optation style.

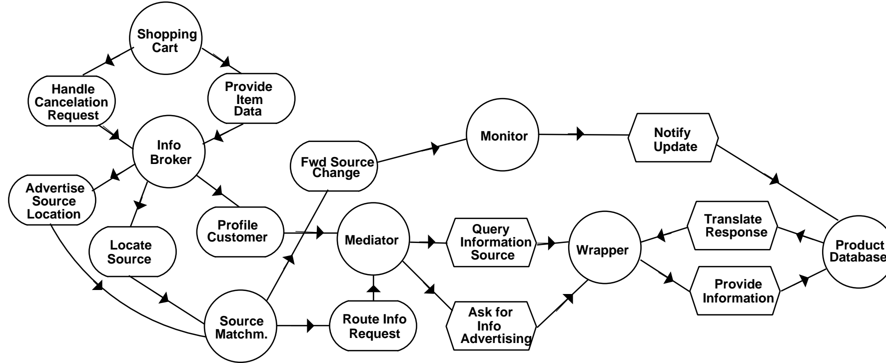

Figure 6. Decomposing the Store Front with Social Patterns.

Social Patterns. A further step in the architectural design of MAS consists of specifying how the goals delegated to each actor are to be fulfilled [15]. For this step, designers can be guided by a catalogue of multi-agent patterns which offer a set of standard solutions. Considerable work has been done in software engineering for defining software patterns (see e.g., [11]). Unfortunately, little emphasis has been put on social and intentional aspects. Moreover, proposals for agent patterns that do address these aspects (see e.g., [1]) are not intended for use at a design level. Instead, such proposals seem to aim at the implementation phase, when issues such as agent communication, information gathering, or connection setup are addressed.

Social patterns [7] are design patterns focusing on social and intentional aspects that are recurrent in multi-agent and cooperative systems. In particular, the structures are inspired by the federated patterns introduced in [13, 15]. We have classified them into two categories.

The *Pair* patterns – such as booking, call-for-proposal, subscription, or bidding – describe direct interactions between negotiating agents. For instance, the **Bidding** pattern involves an initiator and a number of participants. The initiator organizes and leads the bidding process. He publishes the bid to the participants and receives various proposals. At every iteration, the initiator can accept an offer, raise the bid, or cancel the process.

The *Mediation* patterns – such as monitor, broker, matchmaker, mediator, embassy, or wrapper – feature intermediary agents that help other agents to reach an agreement on an exchange of services. For instance, in the **Broker** pattern, the broker agent is an arbiter and intermediary that requests services from a provider to satisfy the request of a consumer.

Figure 6 shows a possible use of the patterns in the e-business system shown in Figure 5. In particular, it shows how to realize the depen-

dencies *Manage catalogue browsing*, *Update Information*, and *Product Information* from the point of view of the Store Front. The Store Front and the dependencies are decomposed into a combination of social patterns involving agents, pattern agents, subgoals and subtasks.

The booking pattern is applied between the *Shopping Cart* and the *Information Broker* to reserve available items. The broker pattern is applied to the *Information Broker*, which satisfies the Shopping Cart 's requests of information by accessing the *Product Database*. The *Source Matchmaker* applies the matchmaker pattern to locate the appropriate source for the *Information Broker*, and the monitor pattern is used to check any possible change in the *Product Database*. Finally, the mediator pattern is applied to dispatch the interactions between the *Information Broker*, the *Source Matchmaker*, and the *Wrapper*, while the wrapper pattern makes the interaction between the *Information Broker* and the *Product Database* possible.

5. Goal Models

Traditional goal analysis consists of decomposing goals into subgoals through an AND- or OR-decomposition. If goal G is AND-decomposed (respectively, OR-decomposed) into subgoals $G_1, G_2, \ldots, G_n$, then all (at least one) of the subgoals must be satisfied for the goal G to be satisfied. Given a goal model consisting of goals and AND/OR relationships among them, and a set of initial labels for some nodes of the graph (S for Satisfied, D for Denied) there is a simple label propagation algorithm which can generate labels for all nodes of the graph [20].

Unfortunately, this simple framework for modeling and analyzing goals won't work for many domains where goals can't be formally defined, and the relationships among them can't be captured by semantically well-defined relations such as AND/OR ones. For example, a goal such as "*Highly reliable system*" has no formal definition which prescribes its meaning, though one may want to define necessary conditions for its fulfillment. Moreover, such a goal may be related to other goals, such as "*Thoroughly debugged system*", "*Thoroughly tested system*" in the sense that the latter obviously contribute to the satisfaction of the former, but this contribution is partial and qualitative. In other words, if the latter goals are satisfied, they certainly contribute towards the satisfaction of the former goal, but don't guarantee it. The framework will also not work in situations where there are contradictory contributions to a goal. For instance, we may want to allow for multiple decompositions of a goal G into sets of subgoals, where some decompositions suggest satisfaction of G while others suggest denial.

Figure 7. A partial goal model for GM.

Tropos proposes a formal model for goals that can cope with qualitative relationships and inconsistencies among goals. Suppose we are modeling the strategic objectives of a US car manufacturer, such as Ford or GM. Examples of such objectives are *increase return on investment* or *increase customer loyalty.* Objectives can be represented as goals, and can be analyzed using goal relationships such as AND, OR, "+" and "−". In addition, we will use "++" (respectively "−−") a binary goal relationship such that if $++(G,G')$ $(--(G,G'))$ then satisfaction of G implies satisfaction (denial) of G'.

For instance, *increase return on investment* may be AND-decomposed into *increase sales volume* and *increase profit per vehicle*. Likewise, *increase sales volume* might be OR-decomposed into *increase consumer appeal* and *expand markets.* This decomposition and refinement of goals can continue until we have goals that are tangible (i.e., someone can satisfy them through an appropriate course of action) and/or observable (i.e., they can be confirmed satisfied/denied by simply observing the application domain).

For vaguely stated goals, such as *increase customer loyalty* we may want to simply model other relevant goals, such as *improve car quality*, *improve car services* and relate them through "+" and "−" relationships, as shown in Figure 7. These goals may influence positively or negatively some of the goals that have already been introduced during the analysis of the goal *increase return on investment*. Such lateral goal relationships may introduce cycles in our goal models.

Examples of observable goals are *Yen rises*, *gas prices rise* etc. When such a goal is satisfied, we will call it an *event* (the kind of event you may read about in a news story) and represent it in our graphical notation as a rectangle (see lower portion of Figure 7).

Figure 7 shows a partial and fictitious goal model for GM focusing on the goal *increase return of investment*. In order to *increase return of investment*, GM has to satisfy both goals *increase sales* and *increase profit per vehicle*. In turn, *increase sales volume* is OR-decomposed into *increase consumer appeal* and *expand markets*, while the goal *increase profit per vehicle* is OR-decomposed into *increase sales price*, *lower production costs*, *increase foreign earnings*, and *increase high margin sales*. Additional decompositions are shown in the figure. For instance, the goal *increase consumer appeal* can be satisfied by satisfying *lower environment impact*, trying to *lower purchase costs*, or reducing the vehicle operating costs (*reduce operating costs*).

The graph shows also lateral relationships among goals. For example, the goal *increase customer loyalty* has positive (+) contributions from goals *lower environment impact*, *improve car quality* and *improve car services*, while it has a negative (−) contribution from *increase sales price*. The root goal *increase return on investment (GM)* is also related with goals concerning others auto manufacturer, such as Toyota and VW. In particular, if GM increases sales, then Toyota loses a share of the North American market; if Toyota increases sales *increase Toyota sales*), it does so at the expense of VW; finally, if VW increases sales (*increase VW sales*), it does so at the expense of GM.

So far, we have assumed that every goal relationship treats S and D in a dual fashion. For instance, if we have $+(G, G')$, then if G is satisfied, G' is partially satisfied, and (dually) if G is denied G' is partially denied. Note however, that sometimes a goal relationship only applies for S (or D). In particular, the $--$ contribution from increase GM sales to increase Toyota sales only applies when increase GM sales is satisfied (if GM hasn't increased sales, this doesn't mean that Toyota has). To capture this kind of relationship, we introduce $-_S$, $-_D$, $+_S$, $+_D$ (see also Figure 7).

In [12] we have presented an axiomatization of a qualitative and a quantitative goal model. We report here the qualitative formalization.

We consider sets of goal nodes G_i and of relations $(G_1, ..., G_n) \stackrel{r}{\longmapsto} G$ over them, including the $(n+1)$-ary relations *and*, *or* and the binary relations $+_S$, $-_S$, $+_D$, $-_D$, $++_S$, $--_S$, $++_D$, $--_D$, $+$, $-$, $++$, $--$. We briefly recall the intuitive meaning of these relations: $G_2 \stackrel{+_S}{\longmapsto} G_1$ [resp. $G_2 \stackrel{++_S}{\longmapsto} G_1$] means that if G_2 is satisfied, then there is some [resp. a full] evidence that G_1 is satisfied, but if G_2 is denied, then nothing is said about the denial of G_1; $G_2 \stackrel{-_S}{\longmapsto} G_1$ [resp. $G_2 \stackrel{--_S}{\longmapsto} G_1$] means that if G_2 is satisfied, then there is some [resp. a full] evidence that G_1 is denied, but if G_2 is denied, then nothing is said about the satisfaction of G_1. The meaning of $+_D$, $-_D$, $++_D$, $--_D$ is dual w.r.t. $+_S$, $-_S$, $++_S$, $--_S$ respectively (by "dual" we mean that we invert satisfiability with deniability). The relations $+$, $-$, $++$, $--$ are such that each $G_2 \stackrel{r}{\longmapsto} G_1$ is a shorthand for the combination of the two corresponding relationships $G_2 \stackrel{r_S}{\longmapsto} G_1$ and $G_2 \stackrel{r_D}{\longmapsto} G_1$. (We call the first kind of relations *symmetric* and the latter two *asymmetric*.)

Let $G_1, G_2, ...$ denote goal labels. We introduce four distinct predicates over goals, $\mathrm{FS}(G)$, $\mathrm{FD}(G)$ and $\mathrm{PS}(G)$, $\mathrm{PD}(G)$, meaning respectively that there is (at least) *full* evidence that goal G is satisfied and that G is denied, and that there is at least *partial* evidence that G is satisfied and that G is denied.

To formalize the propagation of satisfiability and deniability evidence through a goal graph, we introduce in [12] a set of axioms stating: full satisfiability and deniability imply partial satisfiability and deniability, respectively; for an AND relation, full and partial satisfiability of the target node require respectively the full and partial satisfiability of all the source nodes; satisfiability (but not the full satisfiability) propagates through a "$+_S$" relation. Thus, an AND relation propagates the minimum satisfiability value (and the maximum deniability one), while a "$+_S$" relation propagates at most a partial satisfiability value. Dual axioms hold for the other relations.

Given a goal graph, we can perform two different kind of reasoning: *Top-Down* and *Bottom-Up*. In Top-Down reasoning, we concentrate on a set of root goals with a desired assignment (e.g., satisfy all of them), and we want to find an assignment to the leaf nodes consistent with the desired assignment. In other words, we want to find an initial assignment to the leaf nodes that can be propagate the desiderata assignment to the root nodes. In Bottom-Up reasoning, we concentrate on a set of leaf nodes with an initial assignment, and propagate these assignments upwards to find out their implications for root-level goals.

In [12] we have proposed sound and complete algorithms for qualitative and quantitative Top-Down reasoning with goal models. In particular, given a goal model and labels for some of the goals, our algorithms

propagate these labels upwards. If the graph contains loops, propagation proceeds until a fixpoint is reached. We have also developed algorithms for qualitative and quantitative Bottom-Up reasoning.

6. Conclusions

We have presented an overview of the Tropos methodology. The basic assumption that distinguishes our work from others in Requirements Engineering is that actors and goals are used as fundamental concepts for modeling and analysis during all phases of software development, not just early requirements. The distinguishing feature of Tropos compared to other agent-oriented software development methodologies is its emphasis on requirements analysis. Further information about the Tropos project can be found at *www.troposproject.org*.

The methodology has only been applied so far to several modest-size case studies, e.g. [2], with encouraging results. Moreover, the methodology still lacks tools that support the transition between different phases. Another limitation of the methodology is that it has not been used for the development of full-fledged multi-agent systems.

Of course, much remains to be done to further refine and evaluate the proposed methodology. We are currently working on several open problems, such as the development of other formal analysis techniques for Tropos models, and the development of tools that support design activities during different phases of the methodology.

Acknowledgements

We thank all contributors to the Tropos project – in Trento, Toronto and elsewhere – for useful comments, discussions and feedback.

References

1. Y. Aridor and D.B. Lange. Agent design patterns: Elements of agent application design. In *Proc. of the 2nd Int. Conf. on Autonomous Agents, Agents'98*, pages 108–115, St. Paul, USA, May 1998.
2. J. Castro, M. Kolp, and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*. Elsevier, Amsterdam, the Netherlands, (to appear).
3. L.K. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

4. A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, number 2404 in LNCS, Copenhagen (DK), July 2002. Springer.
5. E. Clarke, O.Grumberg, and D.Peled. *Model Checking.* MIT Press, 1999.
6. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.
7. T.T. Do, M. Kolp, and A. Pirotte. Social patterns for designing multi-agent systems. In *Proc. of the 15th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'03*, 2003.
8. A. Fuxman. Formal analysis of early requirements specifications. Master's thesis, University of Toronto, 2001.
9. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements in Tropos: Some experimental results. In *Proceedings of the* 11th *IEEE International Requirements Engineering Conference*, Monterey Bay, California USA, September 2003. ACM-Press.
10. A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA), August 2001. IEEE Computer Society.
11. E. Gamma, R. Helm, J. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
12. P. Giorgini, E. Nicchiarelli, J. Mylopoulos, and R. Sebastiani. Reasoning with Goal Models. In *Proc. Int. Conference of Conceptual Modeling – ER'02*, LNCS, Tampere, Finland, October 2002. Springer.
13. S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multiagent coordination. In *Proc. of the 3rd Int. Conf. on Autonomous Agents, Agents'99*, Seattle, USA, May 1999.
14. N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2), 2000.
15. M. Kolp, P. Giorgini, and J. Mylopoulos. A goal-based organizational perspective on multi-agents architectures. In *Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages, ATAL'01*, Seattle, USA, August 2001.
16. Y. Labrou, T. Finin, and Y. Peng. The current landscape of agent communication languages. *Intelligent Systems*, 14(2):45–52, 1999.
17. A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *IEEE Int. Symposium on Requirements Engineering*, pages 249–261, Toronto (CA), August 2001. IEEE Computer Society.
18. H. Mintzberg. *Structure in fives : designing effective organizations.* Prentice-Hall, 1992.
19. J. Morabito, I. Sack, and A. Bhate. *Organization modeling : innovative architectures for the 21st century.* Prentice Hall, 1999.
20. N. Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw Hill, 1971.
21. W.R. Scott. *Organizations: rational, natural, and open systems.* Prentice Hall, 1998.
22. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
23. E. Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Department of Computer Science, 1995.