Detailing Architectural Design in the Tropos Methodology

Carla T. L. L. Silva [1], Jaelson F. B. Castro [1], John Mylopoulos [2]

[1] Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970, +1 5581
{ctlls,jbc}@cin.ufpe.br
[2] Dept. of Computer Science University of Toronto, 10 King's College Road Toronto M5S3G4, Canada, +1 416 978 5180
jm@cs.toronto.edu

Abstract

Software systems development happens within a context which organizational processes are well-established. Hence, software needs to be built with flexible architectures based in social and intentional concepts to enable software to evolve consistently with its operational environment. In this sense, the Tropos requirements oriented development methodology, has defined a number of organizational architectural styles which are suitable to agent, cooperative, dynamic and distributed applications. In this paper, we use an extended version of UML to describe these novel architectural styles in order to provide a detailed representation of both the structure and behaviour of the architectural design using these styles. This proposal has been applied to an e-commerce software system.

1. Introduction

Companies are continually changing and turning their attention to improve their business strategies. Stakeholders are demanding more flexible and complex systems. Hence, software has to be based on architectures that can evolve and change continually to accommodate new components and meet new requirements. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized for just its initial set of requirements. Tropos [1], a requirements-driven development methodology, has defined organizational architectural styles [6],[7],[8] based on concepts and design alternatives coming from research in organization management, used to model coordination of business stakeholders – individuals, physical or social systems. Tropos relies on the i* notation [4] to describe both requirements and organizational architectural styles. Unfortunately, this notation is not widely accepted by software practitioners nor able to represent some detailed information which sometimes is required in architectural design such as set of signals that are exchanged between architectural components, as well as the valid sequence of these signals (protocol). On the other hand, the Unified Modeling Language – UML [3] has been extended and used to represent the architecture of simple and complex systems. Such an architecture description language is based on UML for Real-Time systems (UML-RT), an UML extension tuned for real time software systems.

In an effort to provide detailed representation in architectural phase of Tropos methodology, as well as to represent the organizational architectural styles into a mainstream industrial notation, in this work we propose to accommodate within UML-RT the concepts and features used for representing organizational architectures into Tropos. In order to validate this proposal, we applied it to an e-commerce software system extracted from [1]. This work is an improvement of another attempt for representing the Tropos concepts in UML [2].

The rest of this paper is organized as follows: Section 2 presents the Tropos methodology. Section 3 describes how software architecture can be modeled using UML. In Section 4, we define how organizational architectures can be modeled using UML-RT. Section 5 depicts the application of the proposal to a case study. Section 6 points to some future work and discusses the contribution of this proposal.

2. The Tropos Methodology

Tropos proposes a software development methodology and a development framework which are founded on concepts used to model early requirements and complements proposals for agent-oriented programming platforms. This methodology is based on the premise that in order to build software that operates within a dynamic environment, one needs to analyze and model explicitly that environment in terms of "actors", their goals

and dependencies on other actors. Tropos supports five phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an organizational setting; the output is an organizational model which includes relevant actors, their goals and dependencies.
- Late requirements, in which the system-to-be is described within its operational environment, along with relevant functions and qualities.
- Architectural design, in which the system's global architecture is defined in terms of subsystems, interconnected through data, control and dependencies.
- Detailed design, in which behaviour of each architectural component is defined in further detail.

In this work, our focus in on architectural design phase. Software architecture is more than just structure, it includes rules on how system functionality is achieved across the structure. Unfortunately, traditional architectural styles for e-business applications [12],[13] focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organizational architecture styles are not described nor the conceptual high-level perspective of the e-business application.

Tropos has defined organizational architectural styles [6],[7],[8] for agent, cooperative, dynamic and distributed applications to guide the design of the system architecture. These architectural styles *(pyramid, joint venture* (Fig. 1), *structure in 5, takeover, arm's length, vertical integration, co-optation, bidding, …*) are based on concepts and design alternatives coming from research on organization management. From this perspective, software system is like a social organization of coordinated autonomous components that interact in order to achieve specific and possibly common goals. The purpose to reduce as much as possible the impedance mismatch between the system and its environment.

For example, the joint venture architectural style (Figure 1) allows a decentralized architecture. The main feature of this style is that it involves an agreement between two or more principal partners/components in order to obtain the benefits derived from operating at a large scale, such as partial investment and lower maintenance costs, as well as reusing the experience and knowledge of the partners/components, since they pursue joint objectives.

To support modeling and analysis during the initial phases, Tropos adopts the concepts offered by *i** [4], 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. This means that both the system's environment and the system itself are seen as organizations of actors, each having goals to be fulfilled and each relying on other actors to help them with goal fulfillment.

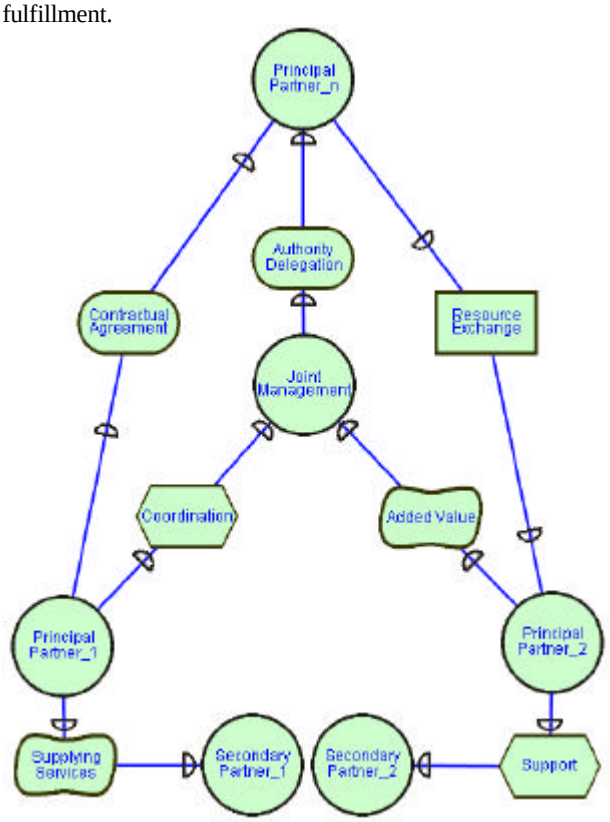

Figure 1. Joint Venture

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. Hence, in Tropos we have the following concepts:

- Actor: An actor is an active entity that carries out actions to achieve goals by exercising its know-how.
- Dependency: A dependency describes an intentional relationship between two actors, i.e., an "agreement" (called dependum) between two actors: the depender and the dependee, where one actor (depender) depends on another actor (dependee) on something (dependum).
- Depender: The depender is the depending actor.
- Dependee: The dependee is the actor who is depended upon.
- Dependum: The dependum is the type of the dependency and describes the nature of the agreement.

- Goal: A goal is a condition or state of affairs in the world that the stakeholders would like to achieve. How the goal is to be achieved is not specified, allowing alternatives to be considered.

- Softgoal: A softgoal is a condition or state of affairs in the world that the actor would like to achieve, but unlike in the concept of (hard) goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to subjective judgment and interpretation of the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal.

- Resource: A resource is an (physical or informational) entity, with which the main concern is whether it is available.

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

The first task during architectural design is to select among alternative architectural styles using as criteria the desired qualities identified in the previous phase (Late Requirements). To this end, the NFR framework [5] can be used to conduct the selection of the most suitable organizational architectural style. More details about the selection and non-functional requirements decomposition process can be found in [6],[7].

In the next section, we show how architectural design can be represented by using an extension of UML. We expose our proposal for representing architectural design in the Tropos methodology using this extension of UML.

3. Architectural Representation in UML

The UMLRT [9],[10] is using UML as an architectural modeling language. Some specific architectural modeling concepts are defined as specializations of generic UML concepts. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints [9]:

- Capsules: A capsule is a stereotype of the UML class concept with some specific features. A capsule uses its ports for all interactions with its environment. The communication with others capsule is done by one or more ports. The interconnection with other capsules is via connectors using signals. A capsule is a specialized active class and is used for modeling a self contained component of a system. For instance, a capsule may be used to capture an entire subsystem, or even a complete system.

- Ports: A port represents an interaction point between a capsule and its environment. They convey signals between the environment and the capsule. The type of signals and the order in which they may appear is defined by the protocol associated with the port. The port notation is shown as a small hollow square symbol. If the port symbol is placed overlapping the boundary of the rectangle symbol denotes a public visibility. If the port is shown inside the rectangle symbol, then the port is hidden and its visibility is private. When viewed from within the capsule, ports can be of two kinds: relay ports and end ports. Relay ports are ports that simply pass all signals through and end ports are the ultimate sources and sinks of all signals sent by capsules. These signals are generated by the state machines of capsules (Figure 8).

- Protocols: A protocol specifies a set of valid behaviors (signal exchanges) between two or more collaborating capsules. However, to make such a dynamic pattern reusable, protocols are decoupled from a particular context of collaborating capsules and are defined instead in terms of abstract entities called protocol roles (stereotype of Classifier Role in UML) (Figure 9).

- Connectors: A connector is an abstraction of a message-passing channel that connects two or more ports. Each connector is typed by a protocol that defines the possible interactions that can take place across that connector (Figure 8).

4. Organizational Architectural Styles In UML

The organizational styles are generic structures defined at a metalevel that can be instantiated to design a specific application architecture. They support non-functional requirements, represented in Tropos methodology such as softgoals, during architectural design phase. Unlike functional requirements which define what a software is expected to do, non-functional requirements specify global constraints on how the software operates or how the functionality is exhibited. NFRs are as important as the functional ones. They are not simply desired quality properties, but critical aspects of dynamic systems without which the applications cannot work and evolve properly. The need to treat non-functional properties explicitly is a critical issue when software architecture is built. Organizational architectures integrate NFR with architectural project, since NFRs are composing part of these styles.

Tropos relies on the i* notation [4] to describe both requirements and represent organizational architectural styles. Unfortunately, this notation is not widely accepted

by software practitioners, since it is just beginning to be recognized as a suitable notation for representing requirements and its tool support is also limited. On the other hand, the Unified Modeling Language [3] has been used to represent the architecture of simple and complex systems. Using UML as an Architecture Design Language in the Tropos methodology allow us for representing detailed information which sometimes is required in architectural design, such as set of signals that are exchanged between architectural components, which are not supported by the i* notation. In the sequel we explain how the concepts of Tropos can be accommodated within UML-RT, in order to represent organizational architectures in UML.

As explained in section 2.1, in Tropos actors are active entities that carries out actions to achieve goals by exercising their know-how. In section 3.1, we explained that in UML-RT, capsules are specialized active classes used for modeling self contained components of a system. Hence, an actor in Tropos is mapped to a capsule in UML-RT (Figure 2). Note that ports are physical parts of the implementation of a capsule that mediate the interaction of the capsule with the outside world.

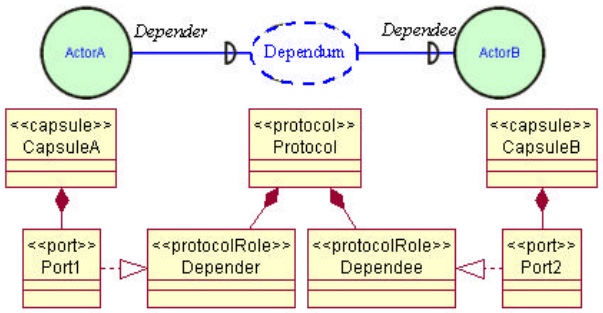

Figure 2. Mapping a dependency between actors to UML

In Tropos a dependency describes an "agreement" (called *dependum*) between two actors playing the roles of *depender* and *dependee*, respectively. The *depender* is the depending actor, and the *dependee*, the actor who is depended upon. Dependencies have the form *depender*⇒*dependum*⇒*dependee*. In UML-RT, a protocol is an explicit specification of the contractual agreement between its participants, which plays specific roles in the protocol. In other words, a protocol captures the contractual obligations that exist between capsules. Hence, a *dependum* is mapped to a protocol and the roles of *depender* and *dependee* are mapped to protocol roles that are comprised by the protocol (Figure 2).

The type of the dependency between two actors (called *dependum*) describes the nature of the agreement. Tropos defines four types of *dependums*: goals, softgoals, tasks and resources. Each type of *dependum* will define different features in the protocol and therefore in ports that realizes its protocol roles. As noted earlier, protocols are defined in terms of entities called *protocol roles*. Since *protocol roles* are abstract classes and ports play a specific role in some protocol, a *protocol role* defines the *type* of a port, which simply means that the port implements the behavior specified by that *protocol role*. As defined earlier, capsules are complex, physical, possibly distributed architectural objects that interact with their surroundings through ports. Note that a port is both a composite part of the structure of the capsule and a constraint on its behavior.

Goal type will be mapped to an attribute with boolean type present into the port that realizes the protocolRole *dependee* (Figure 3). It represents a goal that a capsule is responsible for fulfill by exchanging the signals defined in the protocolRole *dependee*.

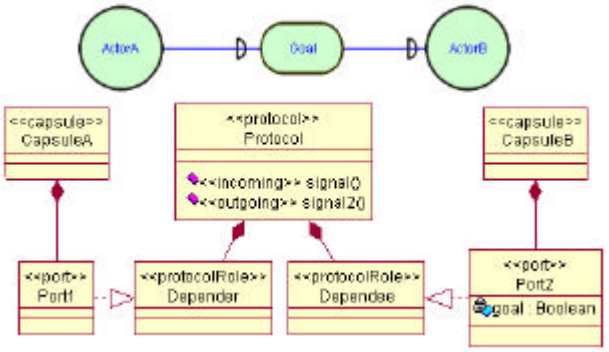

Figure 3. Mapping a goal dependency to UML

Softgoal type is mapped to an attribute with enumerated type present into the port that realizes the protocolRole *dependee* (Figure 4). It represents a quality goal that a capsule is responsible for fulfill to a given extent by exchanging the signals defined in the protocolRole *dependee*.

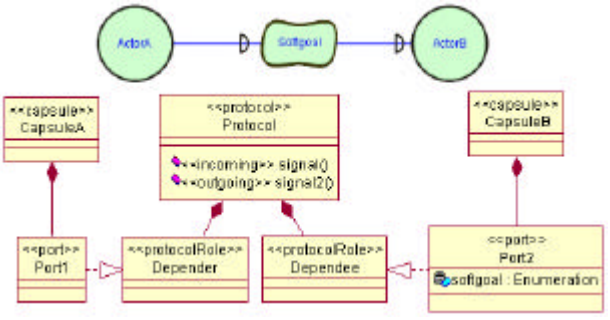

Figure 4. Mapping a softgoal dependency to UML

Resource type is mapped to the return type of an abstract method placed on protocolRole *dependee* that will be realized by a port of a capsule (Figure 5). This return type represents a resource that a capsule is required to provide by exchanging signals defined in the protocolRole *dependee*.

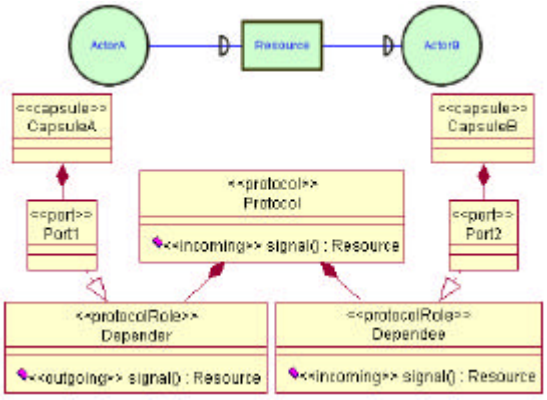

Figure 5. Mapping a resource dependency to UML

Task type is mapped to an abstract method placed on protocolRole *dependee* that will be realized by a port of a capsule (Figure 6). It represents an activity that a capsule is required to perform by exchanging signals defined in the protocolRole *dependee*.

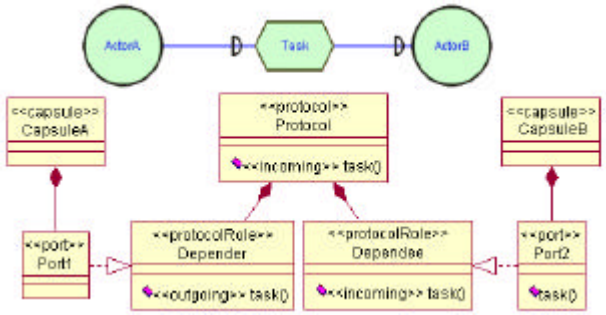

Figure 6. Mapping a task dependency to UML

A more compact form for describing capsules is illustrated in Figure 7, where the ports of a capsule are listed in a special labeled list. The protocol role (type) of a port is normally identified by a pathname since protocol role names are unique only within the scope of a given protocol. However, ports are also depicted in the collaboration diagrams (Figure 8) that describe the internal decomposition of a capsule. In these diagrams, ports are represented by the appropriate classifier roles, i.e., the *port roles*. To reduce visual clutter, port roles are generally shown in iconified form. For the case of binary protocols, an additional stereotype icon can be used: the port playing the conjugate role (*depender* role) is indicated by a white-filled (versus black-filled) square. In that case, the protocol name and the tilde suffix are sufficient to identify the protocol role as the conjugate role; the protocol role name is redundant and should be omitted. Similarly, the use of the protocol name alone on a black square indicates the base role (*dependee* role) of the protocol. In Figure 8, we can see the details of (inside) the capsule and the end port/relay port distinction is indicated graphically.

Figure 7. A capsule class diagram

In UML-RT, each connector is typed by a protocol that specifies the *desired* behavior that can take place over that connector. A key feature of connectors is that they can only interconnect ports that play complementary roles in the protocol associated with the connector. In a class diagram, a connector is modeled by an association while in a capsule collaboration diagram it is declared through an association role. Hence, a dependency (*depender*⇒ *dependum*⇒*dependee*) in Tropos is mapped to a connector in UML-RT (Figure 7 and Figure 8). In the sequel we show how the Joint Venture organizational architectural style (Figure 1) is modeled using UML-RT.

4.1. Joint Venture In UML

The UML-RT notation of capsules, ports and connectors is used to model the architectural actors and their dependencies. In Figure 8, each capsule is representing an actor of the joint venture architecture. When an actor is a *dependee* of some dependency, its corresponding capsule has an implementation port (end port) for each dependency (ex. Port1), which is used to provide services for others capsules. When an actor is a *depender* of some dependency, its corresponding capsule has an implementation port (relay port) to exchange messages (ex. Port3).

The Joint Venture architectural style presents six capsules disposed according to Figure 8. The capsule Joint Management is responsible for ensuring the strategic operation and coordination of such a system and its partner capsules on a global dimension. Through the delegation of authority it coordinates tasks and manages sharing of knowledge and resources. The two secondary partners are capsules responsible for supplying services or for supporting tasks for the organization core. The three principal partners are capsules responsible for managing and controlling themselves on a local

dimension. They can interact directly with other principal partners to exchange, provide and receive services, data and knowledge.

From Figure 1 you can recall the goal dependency *Authority Delegation* between *Principal Partner_n* and *Joint Management* actors. Each actor present in Figure 1 is mapped to a capsule in Figure 8. Each *dependum*, i.e., the "agreement" between these two actors is mapped to the protocol (see Figure 9). A protocol is an explicit specification of the contractual agreement between the participants in the protocol. In our study these participants are the two actors previously mapped to capsules. Each dependency is mapped to a connector in Figure 8. Each connector is typed by the protocol that represents the *dependum* of its corresponding dependency. The type of the dependency describes the nature of the agreement, i.e., the connector type describes the nature of the protocol. The four types of *dependums* (Goal, Softgoal, Task and Resource) are mapped to four types of protocols (Figures 9, 10, 11 and 12).

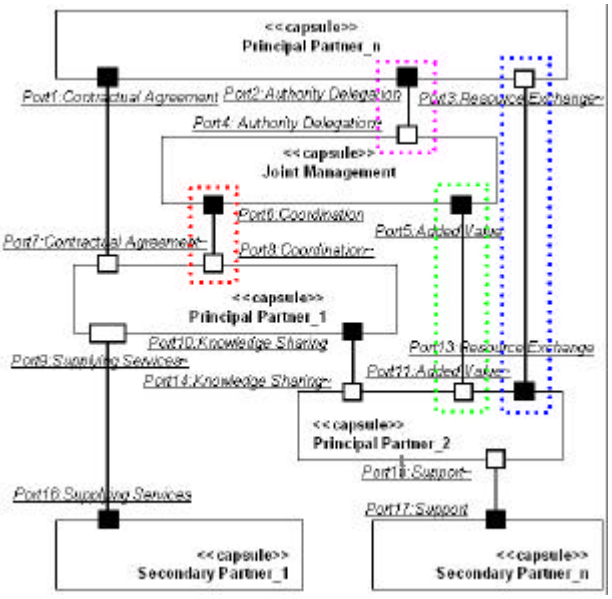

Figure 8. Joint Venture Style in UML-RT's capsule collaboration diagram

For example, in the Goal type, the protocol *Authority Delegation* (Figure 9) assures that this goal will be fulfilled by using the signals described in the protocolRole *dependee*. The goal will be mapped to a boolean attribute present in the port that implements the protocolRole *dependee*. This attribute will be true if the goal has been fulfilled and false otherwise. Hence, in the dependency between *Principal Partner_n* and *Joint Management* capsules depicted in the second doted area of Figure 8, the goal dependency will be mapped to a boolean attribute located in the port which composes the capsule *Principal Partner_n* and implements the protocolRole *dependee* of the protocol that assures the fulfillment of this goal (Figure 9).

Now examine the softgoal dependency *Added Value* between *Principal Partner_2* and *Joint Management* actors depicted in Figure 1. In this case, the protocol *Added Value* (Figure 10) assures that this softgoal will be satisfied in some extent by using the signals described in the protocolRole *dependee*. The softgoal will be mapped to a enumerated attribute present in the port that implements the protocolRole *dependee*. This attribute will represent different degrees of softgoal fulfillment.

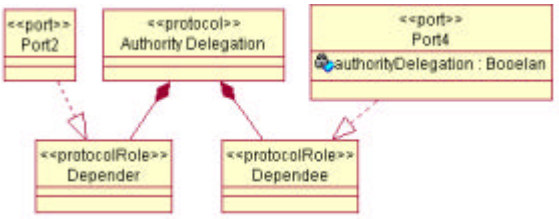

Figure 9. Protocols and Ports representing the Joint Venture's goal dependency Authority Delegation

Hence, in the dependency between *Principal Partner_2* and *Joint Management* capsules depicted in the third doted area of Figure 8, the softgoal dependency will be mapped to a enumerated attribute located in the port which composes the *Joint Management* capsule and implements the protocolRole *dependee* of the protocol that assures some degree of fulfillment of this softgoal (Figure 10).

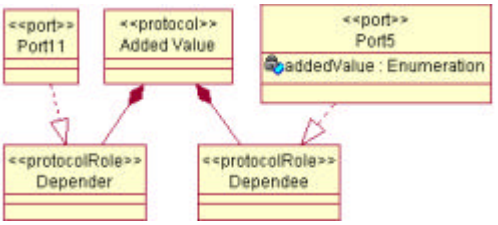

Figure 10. Protocols and Ports representing the Joint Venture's softgoal dependency Added Value

In the sequence, look at the task dependency *Coordination* between *Principal Partner_1* and *Joint Management* actors depicted in the Figure 1. Here, the protocol *Coordination* (Figure 11) assures that this task will be performed by using the signals described in the protocolRole *dependee*. The task itself will be mapped to a <<incoming>> signal in the protocolRole *dependee* and the port that implements that protocolRole will be

committed to realize their signals. Hence, in the dependency between *Principal Partner_1* and *Joint Management* capsules depicted in the first doted area of Figure 8, the task dependency will be mapped to a <<incoming>> signal placed in the protocolRole *dependee* of the protocol that assures the performing of this task. The *Joint Management* capsule is composed by a port which implements this protocolRole *dependee* (Figure 11).

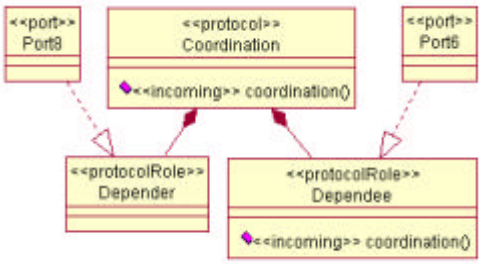

Figure 11. Protocols and Ports representing the Joint Venture's task dependency Coordination

Finally we have the resource dependency *Resource Exchange* between *Principal Partner_2* and *Principal Partner_n* depicted in the Figure 1. Again, the protocol *Resource Exchange* (Figure 12) assures that this resource will be provided by using the signals described as <<incoming>> signals in the protocolRole *dependee.* The resource will be mapped to a <<incoming>> signal that returns an information of type resource in the protocolRole *dependee* and the port that implements that protocolRole will be committed to realize their signals.

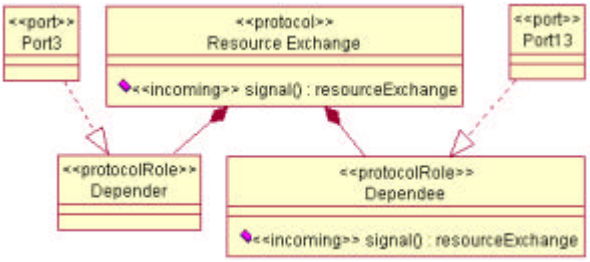

Figure 12. Protocols and Ports representing the Joint Venture's resource dependency Resource Exchange

Hence, in the dependency between *Principal Partner_2* and *Principal Partner_n* capsules depicted in the fourth doted area of Figure 8, the resource dependency will be mapped to an <<incoming>> signal that returns an information of type resource and is placed in the protocolRole *dependee* of the protocol that assures the providing of this resource. The *Principal Partner_2* capsule is composed by a port which implements this protocolRole *dependee* (Figure 12).

Although we have only detailed the mapping of four dependencies in the Joint Venture Style to their respective representation in UML-RT, the remaining ones are mapped analogously, according to their types.

6. Case Study

We extracted a case study from [1] that describes a business organization selling media items (books, newspapers, CDs, etc.) that has decided to open up a B2C retail sales front on the internet named Medi@.

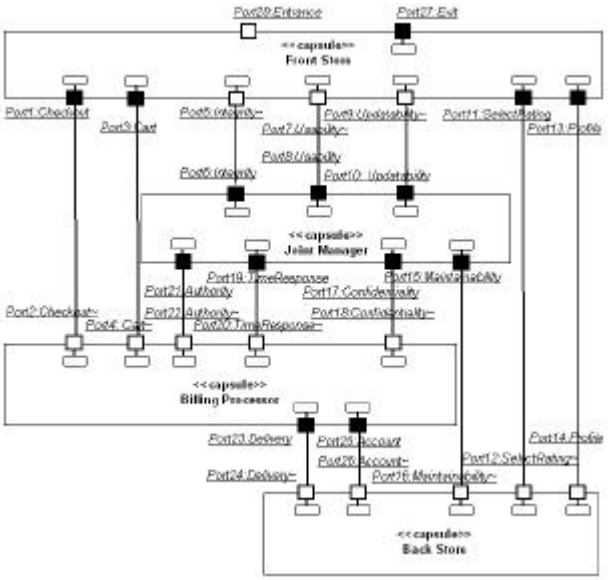

Figure 13 – *Media@* system architecture

Based on the joint venture architectural style, Figure 13 suggests a possible assignment of system responsibilities. *Front Store* primarily interacts with *Customer* and provides her with a usable front-end web application. Moreover, it is responsible for catalogue browsing, items search in database and supplying on-line customers with information about media items. *Back Store* keeps track of all web information about customers, products, sales, bills and other data of strategic importance to *Media Shop*. *Billing Processor* is in charge of the secure management of orders and bills, and other financial data; also of interactions to *Bank Cpy*. *Joint Manager* manages all of the controlling security gaps, availability bottlenecks and adaptability issues, in order to ensure the software non-functional requirements. All four capsules need communicate and collaborate each other in the running system.

Observe that the message exchange between capsules happens in the context defined by protocol implemented by ports that compose each capsule involved in the interaction. For example, the communication protocol in Figure 15 shows a request from Back Store to Front Store for producing the Customer Profile.

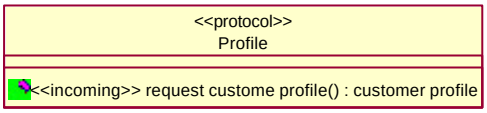

Figure 15. *Profile* Communication protocol between *Front Store* and *Back Store* capsules

Moreover, we can use sequence diagrams to depict the interaction between the capsules which compose the system when realizing a particular scenario: the request for ordering a media item.

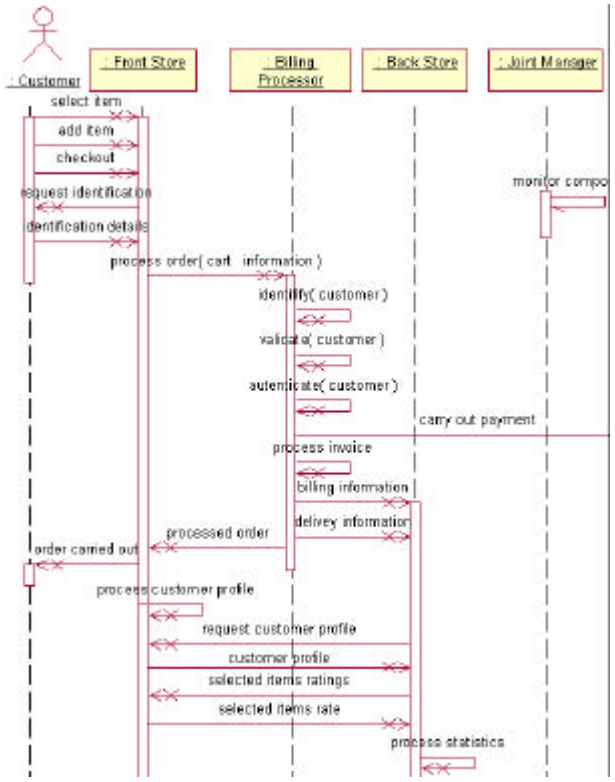

Figure 14. Sequence diagram for *Ordering Media Item* context

Using UML-RT capsules enable us to refine the system architecture to lower-level components (sub-capsules) which depend on each other to realize the whole system responsibilities. Sequence diagrams insert details in architectural behaviour, since it shows the exchanged signals in the interactions, as well as the valid sequence of these signals (communication protocol between capsules).

7. Conclusions and Future Work

In this work, we have been proposed using UML Real-Time to accommodate the concepts and features used for representing organizational architectures in Tropos, nowadays. This proposal has been applied to multi-agent software system development for an e-commerce application. In this paper, we outline an organizational architecture in UML. Our approach is appropriate for:

- Obtaining an architectural model closer to organizational environment where the system will eventually operate, mitigating the existent semantic gap between the software system and its running environment.
- Modeling more detailed architectures both in structural and behavioural aspects.
- Building a flexible architecture with loosely coupled components, which can evolve and change continually to accommodate new components and meet new requirements, as well as support non-functional requirements. Hence, it enables to realize stakeholders' demand for more flexible and complex systems.
- Being able to use UML elements to represent non-UML artifacts enables us to use existing UML toolsets to create those views.
- Making organizational architectures styles widely used in industry, namely by other agent-oriented methodologies or those tuned to open, cooperative, dynamic and distributed systems.

In Tropos, UML is used only in detailed design phase. However using UML-RT for modeling architecture can help Tropos in the following issues:

- Common Representation Model: Modeling information of different types of views (UML and non-UML) can be physically stored in the same repository.
- Unified Way of Cross-Referencing Model Information: Having modeling information stored at one physical location further enables us to cross-reference that information. Cross-referencing is useful for maintaining the traceability among artifacts from architectural design and detailed design phases in Tropos.

To improve this proposal, future work is required to provide systematic guidelines. Currently this processes happens in a ad hoc way based on software engineer experience. Proper guidance will enable us to create instances from architectural metamodels, defined by

Tropos, from requirement models represented in i* notation. Also we intend to model internal behaviour of capsules with state diagram. Moreover, we aim at proposing UML extensions for representing social patterns involving agents, as well as both the structural and behavioural aspects and features defining such a software agents, in the context of Tropos Methodology.

7. References

[1] Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. In Information Systems, Vol. 27. Elsevier, Amsterdam, The Netherlands (2002) 365–389

[2] Mylopoulos, J., Kolp, M., Castro, J.: UML for Agent-Oriented Software Development: the Tropos Proposal. In Proceedings of the Fourth International Conference on the Unified Modeling Language (<<UML>> 2001). Toronto, Canada (2001), LNCS 2185, p.p. 422-423

[3] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language – Reference Manual. Addison Wesley (1999)

[4] Yu., E.: Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995)

[5] Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Publishing (2000)

[6] Kolp, M., Castro, J., Mylopoulos, J.: A social organization perspective on software architectures. In Proc. of the 1st Int. Workshop From Software Requirements to Architectures. STRAW'01, Toronto, Canada (2001) 5–12

[7] Kolp, M., Giorgini, P., Mylopoulos, J.: 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 (2001)

[8] Kolp, M., Mylopoulos, J.: Software architectures as organizational structures. In Proc. ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems", Edmonton, Canada (2001)

[9] Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Rational Whitepaper (www.rational.com) (1998)

[10] OMG: Unified Modeling Language 2.0. Initial submission to OMG RFP ad/00-09-01 (UML 2.0 Infrastructure RFP) and ad/00-09-02 (UML 2.0 Superstructure RFP).: Proposal version 0.63 (draft). http://www.omg.org/.

[11] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, N.J., Prentice Hall (1996)

[12] Conallen, J.: Building Web Applications with UML. Addison-Wesley (2000)

[13] IBM: Patterns for e-business. At http://www.ibm.com/developerworks/patterns (2001)

[14] Silva, C. T. L. L., Castro, J. F. B.: Detailing Architectural Design in the Tropos Methodology. CAISE03 - The 15Th Conference on Advanced Information Systems Engineering, 2003, Klagenfurt/Velden, Austria (to appear).