Modeling Organizational Architectural Styles in UML

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

[1] Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970, + 55 81 32718430
{jbc,ctlls}@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. Today's information systems operate within a dynamic, organizational context and consequently require flexible architectures to ensure that they remain operational and useful. The Tropos software development methodology is founded on the premise that social and intentional concepts (such as those of *actor* and *goal*) can be used throughout the development process from early requirements to implementation. Earlier work within the scope of the project has defined a number of organizational architectural styles which are suitable for cooperative, dynamic and distributed applications. In this paper, we use UML to describe these novel architectural styles. In doing so we are able to provide a detailed representation of both the structure and behaviour of the styles.

1 Introduction

Many information systems fail to properly support the organizations of which they are an integral part. This often happens due to a lack of proper understanding of the organizational context by the system developers. It can also be the result of frequent organizational changes that cannot be accommodated by existing systems (or their maintainers). In this context, requirement engineering has been recognized as the most critical phase in information systems development, because technical considerations have to be balanced against social and organizational ones. The Tropos project [1], [2] has proposed a software development methodology inspired by organizational concepts, which reduces as much as possible this impedance mismatch between an information system and its intended environment. The proposed methodology supersedes traditional development techniques, such as structured and object-oriented ones in the sense that it is tailored to systems which operate within an organizational context.

Companies are continually changing as they strive to improve their business strategies and processes. Stakeholders are demanding ever 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. Software architectures describe a software system at a macroscopic level in terms of a manageable number of subsystems/components/modules interrelated through data and control

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 111–126, 2003.

dependencies. However, an architecture is more than just structure, it includes rules on how system functionality is achieved in terms of the structure. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized specifically for its initial set of requirements. Unfortunately, the classical architectural styles [12] and the styles for e-business applications [13],[14] do not focus on business processes nor on non-functional requirements of the application. As a result, the organizational structure is not described nor the conceptual high-level perspective of the application.

In this context, the Tropos project has defined organizational architectural styles [7],[8],[9] based on concepts and design alternatives coming from research in organization management, used to model coordination of business stakeholders – individuals, physical or social systems. From this perspective, software system is like a social organization of coordinated autonomous components that interact in order to achieve private and shared goals. The NFR framework [6] can be used to conduct the selection of the most suitable organizational architectural style, using as criteria desired qualities (non-functional requirements, or NFRs) identified during requirements analysis. Tropos relies on the i* notation [5] to describe both requirements and 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 tool support is also limited. Moreover, it is not able to represent details that are sometimes required in architectural design, such as the exchange of signals and data between architectural components, as well as the valid sequence of these signals (protocol).

On the other hand, the Unified Modeling Language – UML [4] has been widely accepted as the industry's standard language of blueprints for software. As a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, UML has proven itself valuable in helping organizations to manage the complexity of their systems. UML has also been used to represent the architecture of simple and complex systems. As an architectural description language, UML can provide means for representing design decisions. It can lead to architectural models which describes the high-level design elements of the system and their connectors, supporting different viewpoints of the system under construction. Moreover, it is supported by a wide range of tool providers.

In an effort to provide a detailed representation for the architectural phase of the Tropos methodology, also to represent the organizational architectural styles in terms of a mainstream industrial notation, we propose in this paper an extension of UML to accommodate the concepts and features used for representing organizational architectures in Tropos. Such an extension is based on UML for Real-Time systems, which is tuned for real-time software systems and is being used for modeling software architectures. This proposal is an extension and improvement of an earlier attempt to represent Tropos concepts in UML [3].

The rest of this paper is organized as follows: section 2 presents the Tropos methodology. Section 3 describes how organizational architectural styles can be modeled using UML. In section 4, we discuss related work, while section 5 summarizes the results of this research and outlines future work.

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, Late Requirements, Architectural Design, Detailed Design and Implementation.

Early requirements analysis focuses on the intentions of stakeholders. These intentions are modeled as goals, which through some form of analysis, eventually lead to the functional and non-functional requirements of the system-to-be [15]. Late requirements analysis results in a requirements specification, which describes all functional and non-functional requirements for the system-to-be. In *Tropos*, the information system is represented as one or more actors, 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. Both the process to detect the relevant stakeholders and their goals as well as the method to conduct the transition among Tropos models are out of scope of this paper. For further details about *Tropos*, see [1],[2].

A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. Unfortunately, traditional architectural styles for e-business applications [13],[14] 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 [7],[8],[9] 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.

For example, the joint venture architectural style (Fig. 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** [5], a modeling framework defined in terms of concepts such as *actor* (actors can be *agent*s, *positions* or *role*s), 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.

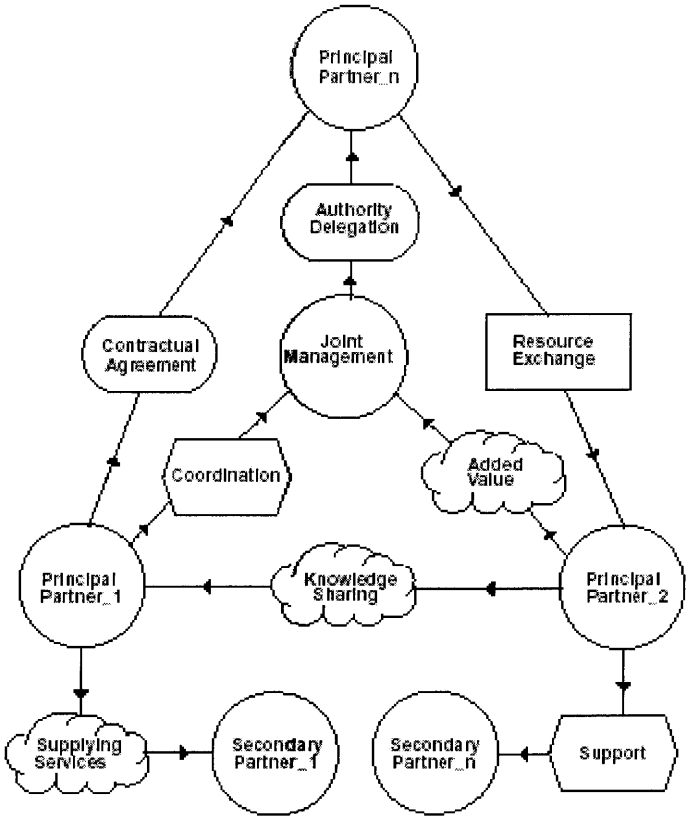

Fig. 1. Joint venture

As shown in Fig. 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*. For further information on the nature of dependencies, please refer to [5]. 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 al-

ternatives to be considered. Goal dependencies are used to represent delegation of responsibility for fulfilling a goal.

- 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. Softgoal dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely (for instance, the appreciation is subjective, or the fulfillment can occur only to a given extent).
- Resource: A resource is an (physical or informational) entity, with which the main concern is whether it is available. Resource dependencies require the dependee to provide a resource to the depender.
- 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. Task dependencies are used in situations where the dependee is required to perform a given activity.

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). They will guide the selection process of the appropriate architectural style. More details about the selection and non-functional requirements decomposition process can be found in [7],[8]. A further step in the architectural design consists in defining how the goals assigned to each actor are fulfilled by agents with respect to social patterns. Further details about social patterns can be found in [1],[2].

The detailed design phase is intended to introduce additional details for each architectural component of a system. In our case, this includes actor communication and actor behavior. To support this phase, we can propose to adopt extensions to UML [3], like AUML, the Agent Unified Modeling Language [18] proposed by the FIPA (Foundation for Physical Intelligent Agents) [16] and the OMG Agent Work group. Tropos also includes techniques for generating an implementation from a detailed design. For further details about these phases of *Tropos* methodology, see [1],[2].

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 Modeling Organizational Architectural Styles in UML

Powerful extensibility mechanisms of UML enables us to represent new concepts in UML and a number of views are captured and sufficiently represented through the use of the UML meta-model. In this section we show how architectural constructs could be derived from more general UML concepts by using these mechanisms and also

describe how the concepts and features used for representing organizational architectures into Tropos are captured and rendered using these constructs.

3.1 Representing Architectures in UML

The UMLRT [10],[11] is using UML as an architectural modeling language. Some specific architectural modeling concepts are defined as specializations of generic UML concepts. This allows us to take advantage of the notation that is widely recognized by software practitioners. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints [10]:

- 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 and end ports. Relay ports (Fig. 8) 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.
- 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) (Fig. 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 (Fig. 8).

3.2 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.

Aiming to narrow the semantic gap between a software architecture and the requirements model from which it is derived, Tropos relies on the i* notation [5] to describe both requirements and 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- UML [4] 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 Real-Time, capsules are specialized active classes used for modeling self contained components of a system. Hence, an actor in Tropos is represented in terms of a capsule in UML-RT (Fig. 2). Note that ports are physical parts of the implementation of a capsule that mediate the interaction of the capsule with the outside world. The motivation and reasoning for mapping these concepts to UML include those of [17].

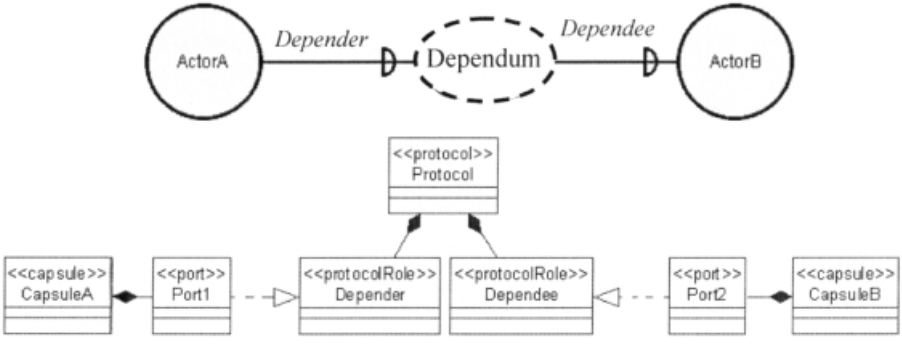

Fig. 2. Mapping a dependency between actors to UML

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 (Fig. 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.

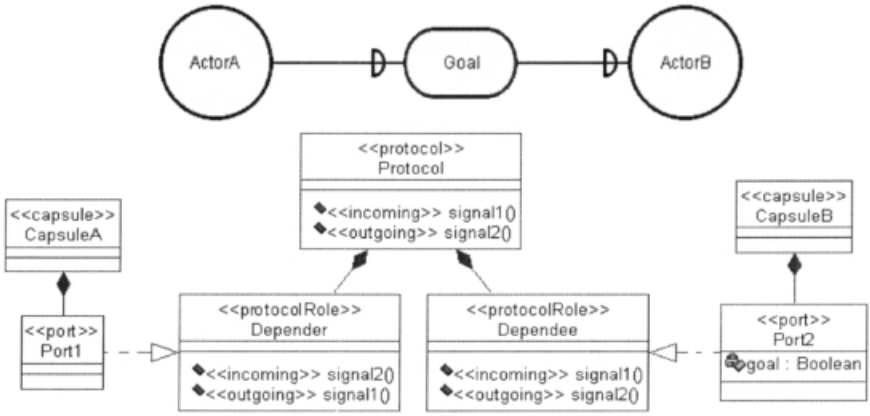

Fig. 3. Mapping a goal dependency to UML

Goal type will be mapped to an attribute with Boolean type present into the port that realizes the protocolRole *dependee* (Fig. 3). The Boolean type is used to depict the goal satisfaction (true) or no satisfaction (false). This attribute represents a goal that a capsule is responsible for fulfill by exchanging the signals defined in the protocolRole *dependee*.

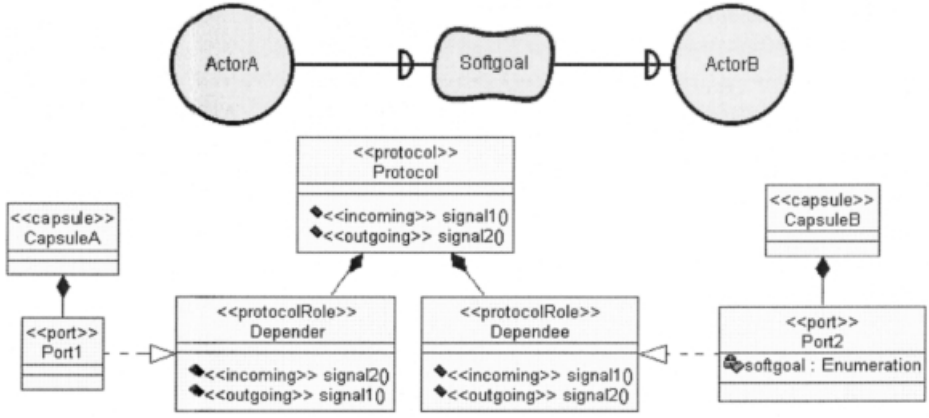

Fig. 4. Mapping a softgoal dependency to UML

Softgoal type is mapped to an attribute with enumerated type present into the port that realizes the protocolRole *dependee* (Fig. 4). The enumerated type is used to depict the degree of satisfaction of the softgoal. This attribute represents a quality goal that a capsule is responsible for fulfill to a given extent by exchanging the signals defined in the protocolRole *dependee*.

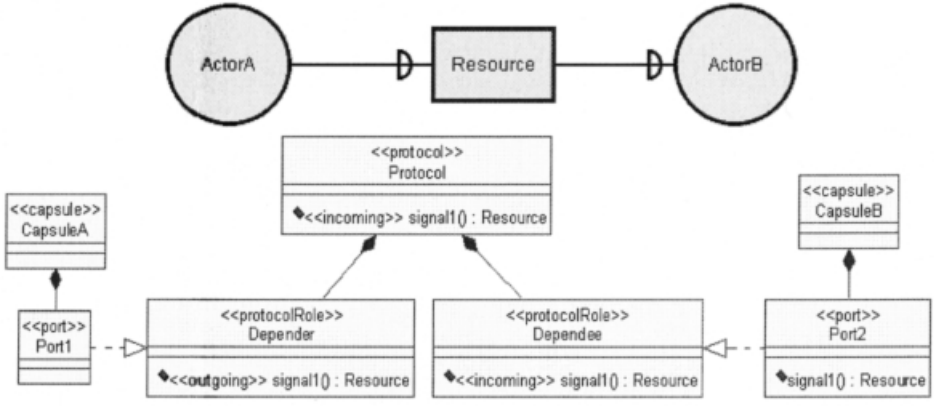

Fig. 5. Mapping a resource dependency to UML

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 (Fig. 5). This return type represents the type of the resulting product from an operation related to some service that the capsule is responsible to perform. This resulting product will represent a resource that a capsule is required to provide by exchanging signals defined in the protocolRole *dependee*.

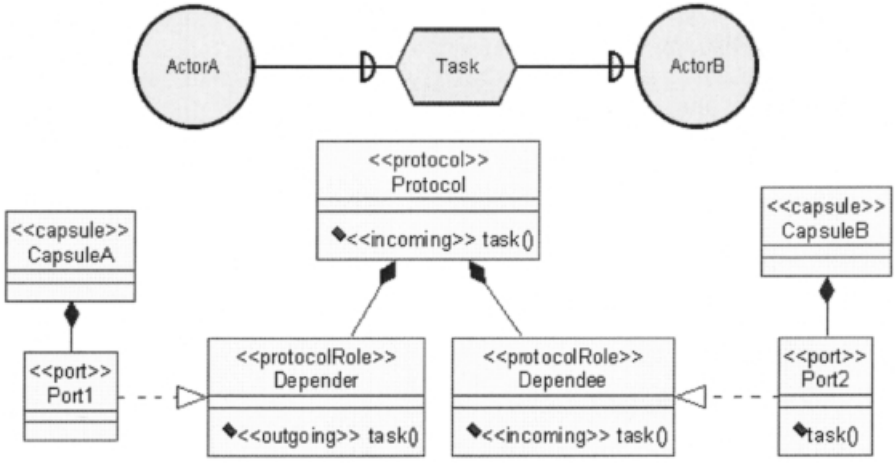

Fig. 6. Mapping a task dependency to UML

Task type is mapped to an abstract method placed on protocolRole *dependee* that will be realized by a port of a capsule (Fig. 6). This method will represent an operation related to some service that the capsule is responsible to perform by exchanging signals defined in the protocolRole *dependee*.

A more compact form for describing capsules is illustrated in Fig. 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 (Fig. 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 Fig. 8, we can see the details of (inside) the capsule and the end port/relay port distinction is indicated graphically.

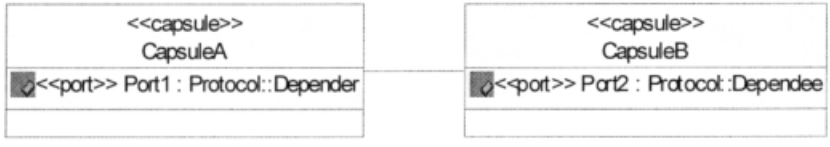

Fig. 7. A capsule class diagram

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 (Fig. 7 and Fig. 8).

In the sequel we show how the Joint Venture organizational architectural style is modeled using UML-RT.

3.3 Joint Venture in UML

The UML notation of capsules, ports and connectors is used to model the architectural actors and their dependencies. In Fig. 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 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 to exchange messages (ex. Port3). However, notice that there is some reasonable distance between the representation given in Figure 1 and that in Figure 8.

This architecture presents six capsules disposed according to Fig. 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 Fig. 1 you can recall the goal dependency *Authority Delegation* between *Principal Partner_n* and *Joint Management* actors. Each actor present in Fig. 1 is mapped to a capsule in Fig. 8. Each *dependum*, i.e., the "agreement" between these two actors is mapped to the protocol in Fig. 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 Fig. 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).

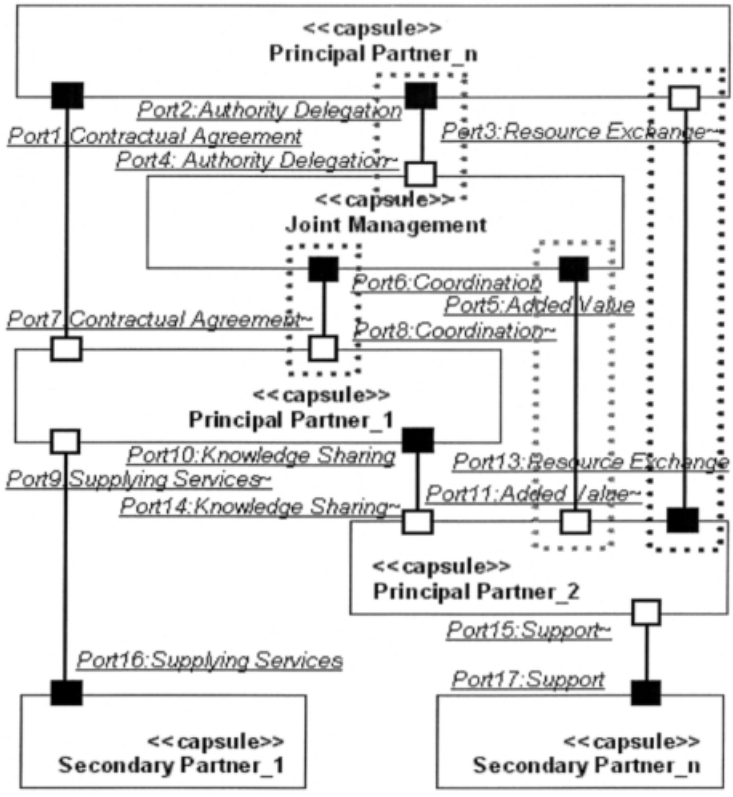

Fig. 8. Joint Venture Style in UML-RT's capsule collaboration diagram

For example, in the Goal type, the protocol *Authority Delegation* (Fig. 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 Fig. 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 (Fig. 9).

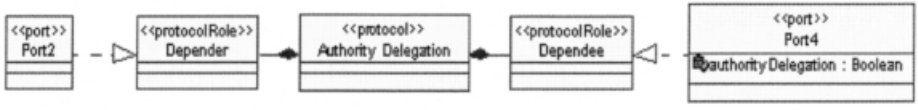

Fig. 9. Protocols and Ports representing the Joint Venture's goal dependency Authority Delegation

Now examine the softgoal dependency *Added Value* between *Principal Partner_2* and *Joint Management* actors depicted in Fig. 1. In this case, the protocol *Added*

Value (Fig. 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. Hence, in the dependency between *Principal Partner_2* and *Joint Management* capsules depicted in the third doted area of Fig. 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 (Fig. 10).

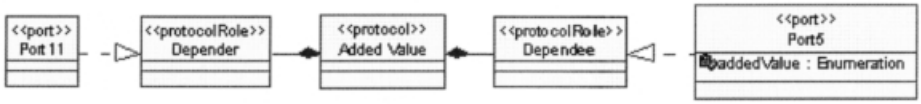

Fig. 10. Protocols and Ports representing the Joint Venture's softgoal dependency Added Value

In the sequence, look at the task dependency *Coordination* between *Principal Partner_1* and *Joint Management* actors depicted in the Fig. 1. Here, the protocol *Coordination* (Fig. 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 Fig. 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* (Fig. 11).

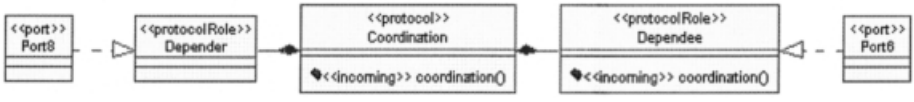

Fig. 11. Protocols and Ports representing the Joint Venture's task dependency coordination

Finally we have the resource dependency *Resource Exchange* between *Principal Partner_2* and *Principal Partner_n* depicted in the Fig. 1. Again, the protocol *Resource Exchange* (Fig. 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. Hence, in the dependency between *Principal Partner_2* and *Principal Partner_n* capsules depicted in the fourth doted area of Fig. 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* (Fig. 12).

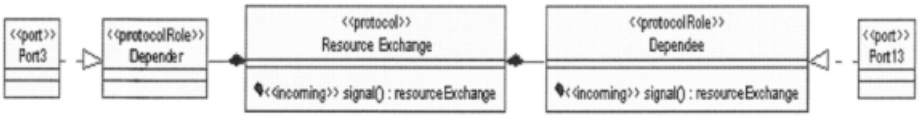

Fig. 12. Protocols and Ports representing the Joint Venture's resource dependency Resource Exchange

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.

4 Related Work

Social concepts have always been a source of inspiration for multi-agent research, and recently the agent community has been returning the favor by exploring the potential of agent-based models for studying social phenomena. The result of this interaction has been the formalization of a number of social and psychological concepts with important applications in engineering agent systems, concepts that are not directly supported in UML. Hence, Parunak [19] addresses an area of agent functionality that goes beyond the capabilities of current UML and presents a number of concepts, including "group", "role", "dependency," and "speech acts," into a coherent syntax for describing organizational structures, and proposes UML conventions and AUML extensions [18] to support their use in the analysis, specif ication, and design of multi-agent systems. In the case of social structures, insights from AALAADIN [20], dependency theory, and holonics can be fused into a single metamodel of groups as composed of agents occupying roles (defined as patterns of dependency and interaction) in an environment. Unfortunately, this approach was not tailored for architecture description.

5 Conclusions and Future Work

This paper proposes a set of UML extensions for representing the organizational architectural styles proposed in [7], based on UML for Real-Time systems. Use of these architectural styles allows us to build flexible architectures, with loosely coupled components, which can evolve and change continually to accommodate new feature requirements. Hence, it enables to realize stakeholders' demand for more flexible and complex systems [9]. Moreover, using organizational architectural styles in UML allow us for representing detailed information such as the communication signals exchanged by the components which compose the architecture. Currently, this additional feature isn't available in architectural design phase of Tropos methodology because it uses the i* notation in this phase. In Tropos, UML is used only in detailed design phase. Further advantages in using UML to model Tropos architectural styles include [17]:

- 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 further develop this proposal, we plan to create a catalogue of organizational architectural styles in UML, also to extend UML to represent social patterns involving agents. Moreover, in order to evaluate our proposal, we intend to apply it to real projects.

References

1. Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. In Information Systems 27(6), Elsevier, Amsterdam, The Netherlands (2002) 365-389
2. Castro, J., Kolp, M., Mylopoulos, J.: Tropos: A requirements-Driven Software Development Methodology. In Proceedings oh the 13th Conf. on Advanced Information Systems Engineering, CAiSE 2001, Interlaken, Switzerland (2001). LNCS 2068, 108-123
3. 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)
4. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language – Reference Manual. Addison Wesley (1999)
5. Yu., E.: Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995)
6. Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Publishing (2000)
7. 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
8. Kolp, M., Giorgini, P., Mylopoulos, J.: A goal-based organizational perspective on multi-agents architectures. In J.J. Ch. Meyer and Milind Tambe (Eds.) Intelligent Agents VIII: Agent Theories, Architectures, and Languages, Springer, August 2002.
9. Kolp, M., Giorgini, P., Mylopoulos, J: Information Systems Development through Social Structures, In Proc. of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ishia, Italy, July 2002.
10. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Rational Whitepaper (www.rational.com) (1998)
11. 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/.
12. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, N.J., Prentice Hall (1996)
13. Conallen, J.: Building Web Applications with UML. Addison-Wesley (2000)
14. IBM: Patterns for e-business. At http://www.ibm.com/developerworks/patterns (2001)
15. Dardenne, A., Lamsweerde, A.V., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming, Vol. 20 (1993) 3–50
16. FIPA: The Foundation for Intelligent Physical Agents. At http://www.fipa.org (2001)

17. Medvidovic, N., Rosenblum, D.S., Robbins, J.E., Redmiles D.F.: Modeling Software Architectures in the Unified Modeling Language. Computer Science Department, University of Southern California, Los Angeles (2000)
18. Bauer, B., Muller, J., Odell., J.: Agent UML: A formalism for specifying multiagent interaction. In Proc. of the 1st Int. Workshop on Agent-Oriented Software Engineering. AOSE'00, Limerick, Ireland (2001) 91–104
19. Parunak, H.V.D., Odell., J.: Representing Social Structures in UML. Proc. of the Agent-Oriented Soft. Engineering Workshop. Agents 2001 Conference, Montreal, Canada (2001)
20. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In Proceedings of Third International Conference on Multi-Agent Systems. ICMAS'98, IEEE Computer Society (1998) 128-135