

INFORMATION INTEGRATION ARCHITECTURE DEVELOPMENT: A MULTI-AGENT APPROACH

Stéphane Faulkner, Manuel Kolp, Tai Nguyen, Adrien Coyette, Tung Do
*Information Systems Research Unit, University of Louvain,
1 Place des Doyens, 1348 Louvain-la-Neuve, Belgium
Email: {faulkner, kolp, nguyen, coyette, do}@isys.ucl.ac.be*

Abstract. Multi-Agent Systems (MAS) architectures are gaining popularity for building open, distributed, and evolving software required by systems such as information integration applications. Unfortunately, despite considerable work in software architecture during the last decade, few research efforts have aimed at truly defining patterns and languages for designing such multi-agent architectures. We propose a modern approach based on organizational structures and architectural description languages to define and specify multi-agent architectures notably in the case of information integration system design as illustrated in this paper.

1 INTRODUCTION

Architectures for integrating information extracted from multiple heterogeneous sources allow to effectively exploit the numerous sources available on-line through the World Wide Web. Such architectures permit users to access and query numerous information sources to obtain an integrated answer. The sources may be conventional databases or other types of information, such as collections of Web pages.

Designing information integration systems can rapidly become complex. Indeed, such processes require software architecture to operate within distributed environments that must evolve over time to cope with the dynamics and heterogeneity of information sources.

Not surprisingly, researchers have been looking for new software designs that cope with such requirements. One promising source of ideas that has been considered in recent years for designing such information integration software is the area of Multi-Agent System (MAS) architectures. They appear to be more flexible, modular and robust than traditional including object-oriented ones. They tend to be open and dynamic in the sense they exist in a changing organizational and operational environment where new components can be added, modified or removed at any time.

To cope with the ever-increasing complexity of the design of software architecture, architectural design has received through the last decade increasing attention as an important field of software engineering.

Practitioners have come to realize that getting an architecture right is a critical success factor for system life-cycle and have recognized the value of making explicit architectural descriptions and choices in the development of new software.

To this end, a number of architectural description languages (ADL) [2] and architectural styles [5] have been proposed for representing and analyzing architectural designs. An *architectural description language* provides a concrete syntax for specifying architectural abstractions in a descriptive notation while an *architectural style* constitutes an intellectually manageable abstraction of system structure that describes how system components interact and work together.

Unfortunately, despite this considerable work, few research efforts have aimed at truly defining styles and description languages for agent architectural design. To fill this gap, we have defined, in the SKwyRL¹ project, architectural styles for multi-agent systems based on an organizational perspective [3] and have proposed in [4] SKwyRL-ADL, an agent architectural description language. This paper continues and integrates this research: it focuses on a multi-agent perspective for designing and specifying information integration architecture based on organizational styles and SKwyRL-ADL. The joint-venture organizational style will be instantiated to design the architecture of the system and the specifications will be expressed in a formal way with SKwyRL-ADL.

The rest of the paper is organized as follows. Section 2 introduces some perspectives of SKwyRL insisting on the BDI model, our ADL and organizational styles. Section 3 describes our multi-agent approach on information integration system development, including

¹ Socio-Intentional Architecture for Knowledge Systems and Requirements Elicitation (<http://www.isys.ucl.ac.be/skwyrl/>)

the design of the global architecture with organizational styles, its formal specification with SKwyRL-ADL and the corresponding implementation on an agent-oriented platform. Finally, Section 4 concludes the research.

2 ADL AND STYLES IN SKWYRL

We have detailed in the SKwyRL project an agent ADL called SKwyRL-ADL [4] that proposes a set of abstractions that are fundamental to the description and specification of agent architectures based on the BDI (Belief-Desire-Intention) agent model. To help the reader to understand our ADL specification in the rest of the paper, we briefly present the main elements of SKwyRL-ADL including the BDI agent model. SKwyRL-ADL is composed of two sub-models which operate at two different levels of abstraction: *internal* and *global*. The internal model captures the states of an agent and its potential behavior. The global model describes the interaction among agents that compose the multi-agent architecture. We will also introduce organizational styles through the description of one of them, the joint venture, that will be used later on in the paper.

2.1 The BDI Agent Model

An *agent* defines a system entity, situated in some environment that is capable of flexible autonomous action in order to meet its design objective [9].

An agent can be useful as a stand-alone entity that delegates particular tasks on behalf of a user. However, in the overwhelming majority of cases, agents exist in an environment that contains other agents. Such environment is a agent system that can be defined as an *organization* composed of autonomous and proactive agents that interact with each other to achieve common or private goals [7].

In order to reason about themselves and act in an autonomous way, agents are usually built on rationale models and reasoning strategies that have roots in various disciplines including artificial intelligence, cognitive science, psychology or philosophy. An exhaustive evaluation of these models would be out of the scope of this paper or even this research work. A simple yet powerful and mature model coming from cognitive science and philosophy that has received a great deal of attention, notably in artificial intelligence, is the Belief-Desire-Intention (BDI) model [1]. This approach has been intensively used to study the design of rationale agents

and is proposed as a keystone model in numerous agent-oriented development environments such as JACK [6]. The main concepts of the BDI agent model are in addition to the notion of agent itself we have just explained:

- *Beliefs* that represent the informational state of a BDI agent, that is, what it knows about itself and the world;
- *Desires (or goals)* that are its motivational state, that is, what the agent is trying to achieve;
- *Intentions* that represent the deliberative state of the agent, that is, which plans the agent has chosen for possible execution.

2.2 Internal Model

Figure 1 illustrates the main entities and relationships of the internal model of SKwyRL-ADL. The agent needs knowledge about the environment in order to reach decisions. Knowledge is contained in agents in the form of one of many *knowledge bases*. A Knowledge base consists of a set of *beliefs* that the agent has about the environment and a set of *goals* that it pursues. A belief represents a view of the current environment states of an agent. However, beliefs about the current state of the environment are not always enough to decide what to do. In other words, as well as a current state description, the agent needs some goal information, which describes an environment state that is (not) desirable.

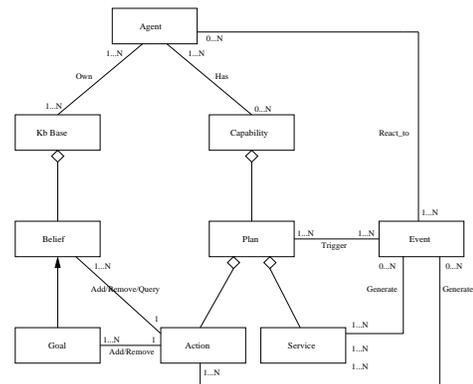


Figure 1: Conceptual Representation of the Internal Model

The intentional behavior of an agent is represented by its *capabilities* to react to *events*. An event is generated either by an *action* that modifies beliefs or adds new goals, or by services provided from another agent. Note that these services are represented in the global model because they involve interaction among agents that compose the agent system.

An event may invoke (trigger) one or more *plans*; the agent commits to execute one of them, that is, it becomes intention. A plan defines the sequence of action to be chosen by the agent to accomplish a task or achieve a goal. An action can query or change the beliefs, generate new events or submit new goals.

2.3 Global Model

Figure 2 conceptualizes the global model which describes the interaction among agents that compose the agent system.

Configurations are the central concept of architectural design, consisting of an interconnected set of *agents*. The topology of a configuration is defined by a set of bindings between provided and required services.

An agent interacts with its environment through an interface composed of sensors and *effectors*. An effector provides to the environment a set of services. Then, a sensor requires a set of services from the environment. A service is an action involving an interaction among agents.

The whole agent system is specified with an *architecture* which contains a set of configurations. An architecture represents the whole system by one or more detailed configuration descriptions.

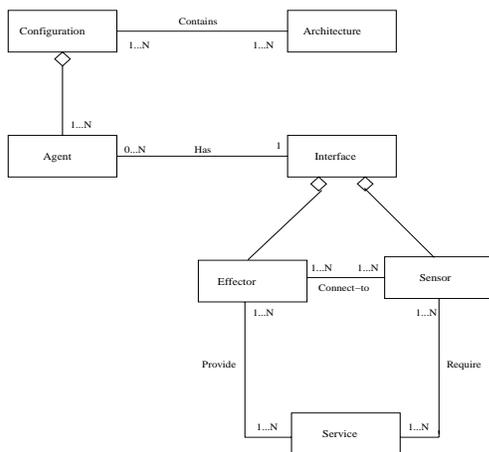


Figure 2: Conceptual Representation of the Global Model

2.4 Multi-Agent Architectural Styles

A key aspect to conduct architectural design in SKwyRL is the specification and use of *organizational styles* (see e.g., [4, 7]) These are socially-based design alternatives inspired by models and concepts from organizational

theories that analyze the structure and design of real-world human organizations. These are the structure-in-5, the joint venture, the chain-of-values, the matrix, the takeover, ...

For instance, the multi-agent architecture we propose in Figure 3 has been designed following and adapting the joint-venture organizational style detailed in [4]. In a few words, the joint-venture organizational style is a meta-structure that defines an organizational system that involves agreement between two or more partners to obtain mutual advantages (greater scale, a partial investment and to lower maintenance costs...). A common actor, the *joint manager*, assumes two roles: a *private interface role* to coordinate partners of the alliance, and a *public interface role* to take strategic decisions, define policy for the private interface, represent the interests of the whole partnership with respect to external stakeholders and ensure communication with the external actors. Each partner can control itself on a local dimension and interact directly with others to exchange resources, data and knowledge.

3 MAS Architecture for Information Integration

GOSIS² is a typical information integration application we have developed using the architectural concepts explained in Section 2. The application provides a Multi-Agent System architecture to support the integration of information coming from different heterogeneous sources.

This section explains how we have used SKwyRL-ADL to formally specify each architectural aspect (belief, goal, plan, action, interface, configuration, service ...) of the application.

3.1 GOSIS Architecture

Figure 3 models the architecture of GOSIS using the *i** model [8] following the joint-venture organizational style we have introduced in Section 2. *i** is a graph, where each node represents an *actor* (or system component) and each link between two actors indicates that one actor depends on the other for some goal to be attained. A dependency describes an “agreement” (called *dependum*) between two actors: the *dependor* and the *dependee*. The *dependor* is the depending actor, and the *dependee*, the actor who is depended upon. The type of the dependency describes the nature of the agreement. *Goal* dependencies represent delegation of responsibility for fulfilling a goal; *softgoal*

² aGent-Oriented Source Integration System

dependencies are similar to goal dependencies, but their fulfilment cannot be defined precisely; task dependencies are used in situations where the dependee is required.

As show in Figure 3, actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; dependencies have the form *dependum* → *dependee*.

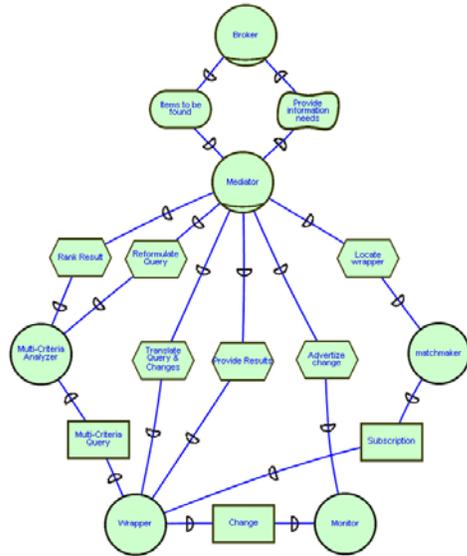


Figure3: The GOSIS Architecture in joint-venture

Figure 3 shows that the mediator plays the role of the joint manager private interface, other joint venture partners are the wrapper, the monitor, the matchmaker and the multi-criteria analyzer. The public interface is assumed by the broker.

When a user wishes to send a request, it contacts the broker agent, which serves as an intermediary to select one or more mediator(s) that can satisfy the user information needs. Then, the selected mediator(s) decomposes the user’s query into one or more sub-queries regarding the appropriate information sources, eventually compiles and synthesizes results from the source and returns the final result to the broker.

When the mediator identifies repetitively the same user information needs, this information of interest is extracted from each source, merged with relevant information from the other sources, and stored as knowledge by the mediator. Each stored knowledge constitutes a materialized view the mediator has to maintain up-to-date.

A wrapper and a monitor agents are connected to each information source. The wrapper ensures two roles. It has to translate the sub-query issued by the mediator in

the native format of the source and translate the source response in the data model used by the mediator.

The monitor is responsible for detecting changes of interest (e.g., a change which affects a materialized view) in the information source and for reporting them to the mediator. Changes are then translated by the wrapper and sent to the mediator.

It may also be necessary for the mediator to obtain information concerning the localization of a source and its connected wrapper able to provide current or future relevant information. This kind of information is provided by the matchmaker agent, which lets the mediator directly interact with the correspondent wrapper. The matchmaker plays the role of a “yellow-page” agent. Each wrapper advertises its capabilities by subscribing to the yellow page agent. The wrapper that no longer wishes to be advertised can request to be unsubscribed.

Finally, the multi-criteria analyzer reformulates a sub-query (sent by a mediator to a wrapper) through a set of criteria in order to express the user preferences in a more detailed way, and refines the possible domain of results.

3.2 GOSIS Formal Specification

The architecture described in Figure 3 gives an organizational representation of the system-to-be including relevant actors and their respective goals, tasks and resource inter-dependencies. This model can serve as a basis to understand and discuss the assignment of system functionalities but it is not adequate to provide a precise specification of the system details. As introduced in Section 2, SKwyRL-ADL provides a set of formal agent-oriented constructors that allows to detail in a formal and consistent way the software architecture as well as its agent components and their behaviours.

Figure 4 shows a high-level formal description of the *Mediator* agent. Three aspects of this agent component are of concern here: the *interface* representing the interactions in which the agent will participate, the *knowledge base* defining the agent knowledge capacity and the *capabilities* defining agent behaviors.

SkwyRL-ADL allows to work at different levels of architectural abstractions (i.e., different views of the system architecture) to encapsulate different components of the system in independent hierarchical descriptions. For instance, in Figure 4 the *Mediator* agent has a set of knowledge bases (KB) and a set of capabilities (CP), but the description level chosen here does not specify the

details of the beliefs composing the KB or the plans and events composing each capability.

The rest of the section focuses on the *Mediator* agent to give an example of a refinement specification with our ADL for each of the three aspects of the agent: interface, KB and capabilities.

```

Agent:{ Mediator
  Interface:
    Sensor[require(query_translation)]
    Sensor[require(query reformulation)]
    Sensor[require(change_advertizings)]
    ...
    Effector[provide(founded_items)]
  KnowledgeBase:
    Results_KB
    MatchMaker_Info_KB
    DataManagement_KB
    ...
  Capabilities:
    Handle_Request_CP
    Materialized_Views_CP
    Wrapper_Localizaion_CP
    ... }

```

Figure 4: Agent Structure Description of the Mediator

Interface. The agent *interface* consists of a number of effectors and sensors for the agent. Each of them represents an action in which the agent will participate. Each effector provides a service that is available to other agents, and each sensor requires a service provided by another agent. The correspondence between a required and a provided service defines an interaction. For example, the Mediator needs the *query_translation* service that the Wrapper provides.

Such interface definition points two aspects of an agent. Firstly, it indicates the expectations the agent has about the agents with which it interacts. Secondly, it reveals that the interaction relationships are a central issue of the architectural description. Such relationships are not only part of the specification of the agent behavior but reflect the potential patterns of communication that characterize the ways the system reason about itself.

The required query translation service is described in greater detail in figure 5. We can see that the mediator (sender) initiates the service by asking the wrapper (receiver) to translate a query. To this end, the mediator provides to the wrapper a set of parameters allowing to define the contents of this query. Such mediator query is specified as belief with the predicate search and the following terms:

```
search(RequestType,ProductType(+),FilteredKeyword(+))
```

Each term represents, respectively, the type of the query (normal advanced in the case of multi-criteria refinement), the type of product and one or many keywords that must be included in or excluded from the results.

```

Service:{Ask(query_translation)
  sender: Mediator
  parameters: rt:RequestType ^ pt:ProductType ^
              fk(+):FilteredKeyword
  receiver: Wrapper
  Effect:Add(Translation_Management_KB, search(rt,pt,fk(+))

```

Figure 5: A Service Specification

The service effect indicates that a new search belief is added to the Translation_Management KB of the wrapper.

Knowledge Bases. A *knowledge base* (KB) is specified with a name, a body and a type. The name identifies the KB whenever an agent wants to query or modify them (add or remove a belief). The body represents a set of beliefs in the manner of a relational database schema. It describes the beliefs the agent may have in terms of fields. When the agent acquires a new belief, values for each of its fields are specified and the belief is added to the appropriate KB as a new tuple. The *KB type* describes the kind of formal knowledge used by the agent. A *Closed world* assumes that the agent is operating in a world where every tuple it can express is included in a KB at all times as being true or false. Inversely, in an *open world* KB, any tuple not included as true or false is assumed to be unknown. Figure 6 specifies the Translation_Management_KB:

```

KnowledgeBase: {Translation_Management_KB
KB_body:
  search(RequestType,ProductType,FilteredKeyword(+))
  source_resource(InfoType(+))
  source_modeling(SourceType,Relation(+),Attributes(+))
  dictionary(MediatorTerm,SourceType,Correspondence)
KB_type: closed_world }

```

Figure 6: A Knowledge Base Specification

The '+' symbol means that the attribute is multi-valued.

Capabilities formalize the behavioral elements of an agent. It is composed of plans and events that together define the agent's abilities. It can also be composed of

sub-capabilities that can be combined to provide complex behavior.

Figure 7 shows the *Handle_Request* capability of the *Mediator* agent. The body contains the plans the capability can execute and the events it can post to be handled by other plans or can send to other agents. For example, the *Handle_Request* capability is composed of two plans: *DecompNmlRq* is used to decompose a normal request, *DecompMCRq* to decompose a multi-criteria request.

```

Capability: { Handle_Request_CP
  CP_body:
    Plan DecompNmlRq
    Plan DecompMCRq
    SendEvent FailUserRq
    SendEvent FailDecompMCRq
    PostEvent ReadyToHandleRst }

```

Figure 7: A Capability Specification

A plan defines the sequence of actions and/or services (i.e., actions that involve interaction with other agents) the agent selects to accomplish a task or achieve a goal. A plan consists of:

- an *invocation condition* detailing the circumstances, in terms of beliefs or goals, that cause the plan to be triggered;
- an *optional context* that defines the preconditions of the plan, i.e., what must be believed by the agent for a plan to be selected for execution;
- the *plan body*, that specifies either the sequence of formulae that the agent needs to perform, a formula being either an action or a service to be executed;
- an *end state* that defines the post-conditions under which the plan succeeds;
- and optionally a set of services or actions that specify what happens when a *plan fails* or *succeeds*.

Figure 8 specifies the *DecompNmlRq* plan that decomposes a normal request.

The supplementary condition about the existence of a *materialized_view* belief is specified by the context. The context is used in the selection of the most appropriate plan in a given situation. When the plan specification does not define a context, the plan is selected to be executed only based on the invocation condition.

As soon as the invocation condition and the context are true, the sequence of actions or services specified in the plan body can be executed. The *DecompNmlRq* plan body is composed by an action sequence and a service. The mediator selects from the

wrapper beliefs one or many wrappers ($wp(+)$) able to translate the decomposed sub-queries. A translation service (*Ask(query_translation)*) is then selected from the selected wrappers.

```

Plan:{ DecompNmlRq
  invoc:
    dd(Request_KB, user_keyword( $pt(+)$ ), $kw(+)$ )
  with
     $pt$ :ProductType From Mediator.Ask(user_info-needs).reply_with//
  context:
     $\neg$  materialized_view(ProductType =  $pt(+)$ ),Keyword =  $kw(+)$ )
  body:
     $\forall$   $pt$  : ProductType  $\in$  user_keyword( $pt(+)$ ), $kw(+)$ ) DO

    action select_wrapper(wrapper(WrapperLocalization,
      TranslationService(+))
  as  $wp(+)$ : Wrapper
  service:{Ask(query_translation)
    sender: Mediator
    parameters:  $rt$ :RequestType  $\wedge$   $pt$ :ProductType  $\wedge$ 
       $kw(+)$ :Keyword
    receiver:  $wp(+)$ : Wrapper
  effect: Add(Translation_Management_KB, search( $rt$ , $pt$ , $kw(+)$ ))
  End-DO
  endstate:
     $\forall$   $pt$  : ProductType  $\in$  user_keyword( $pt(+)$ ), $kw(+)$ )
    Add(Translation_Management_KB, search( $rt$ , $pt$ , $fk(+)$ ))
  succeed:
    count(search( $rt$ , $pt$ , $kw(+)$ ))
  effect: Add(Request_Kb, old_user_keyword( $pt$ , $kw(+)$ )) }

```

Figure 8: A Plan Specification

The plan succeeds when the *endstate* statement is or become true. Moreover, SkwyRL-ADL also specifies what happens when a plan reaches its endstate or fails, by considering Further courses of action or service can also be specified to consider what happens next when the plan succeeds or fails. For example, the succeed specification for *DecompNmlRq* counts the number of executions of the current sub-query to identify a potential new materialized view.

Configuration To describe the complete topology of the system architecture, the agents of an architectural description are combined into a SKwyRL *configuration*.

Instances of each agent or service that appear in the configuration must be identified with an explicit and unique name.

The configuration also describes the collaborations (i.e., which agent participates in which interaction) through a one-to-many mapping between provided and required service instances.

Part of the GOSIS configuration with instance declarations and collaborations is given in Figure 9.

“(min)...(max)”. indicates the smallest acceptable integer, and the largest. An omitted cardinality (as is the case with (max) in the broker, mediator and wrapper agents), means no limitation.

```

Configuration GOSIS
Agent Broker[nb: 1...]
Agent Mediator[nm: 1...]
Agent Wrapper[nw: 1...nS] with nS = number of
    information sources
Agent Monitor[nmo: 1...nS]
Agent Matchmaker
Agent Multi-Criteria-analyzer
Service Tell(query_translation)
Service Ask(query_translation)
Service Achieve(result)
Service Do(result)
....
Instances
BRnb: Broker MEnm: Mediator
WRnw: Wrapper
MOnmo: Monitor
MA: Matchmaker
MCA: Multi-Criteria-Analyzer
Tellquerytrans: Tell(query_translation)
Askquerytrans: Ask(query_translation)
Achres: Achieve(result)
Dores: Do(result)
....

Collaborations
MEnm.Askquerytrans --- Tellquerytrans.WRnw;
MEnm.Achres --- Tellres.WRnw;
MEnm.Asksubs --- Tellsubs.MA;
....
End GOSIS

```

Figure 9: The GOSIS Parameterized Configuration

Such a configuration allows for dynamic reconfiguration and architecture resolvability at run-time. Configurations separate the description of composite structures from the description of the elements that form those compositions. This permits reasoning about the composition as a whole and to reconfigure it without having to examine each component of the system

4 CONCLUSION

Nowadays, software engineering for new enterprise application domains such as data integration is forced to build up open systems able to cope with distributed, heterogeneous, and dynamic information issues. Most of these software systems exist in a changing organizational and operational environment where new components can be added, modified or removed at any time. For these reasons and more, multi-agent systems architectures are gaining popularity in that they do allow dynamic and evolving structures which can change at run-time.

Architectural design has received considerable attention for the past decade which has resulted in a collection of well-understood architectural styles and formal architectural description languages. Unfortunately, these works have focused object-oriented rather than agent-oriented systems. This paper has described an approach based on organizational styles and an agent architectural description language we have defined to design multi-agent systems architectures in the context of information integration system engineering. The paper has proposed a validation of the approach: it has been applied to develop GOSIS, an information integration platform implemented on the JACK agent development environment.

REFERENCES

- [1] M. E. Bratman. Intention, Plans and Practical Reason. Harvard University Press, 1987.
- [2] P. C. Clements. A Survey of Architecture Description Languages. In Proc. of the Eighth International Workshop on Software Specification and Design, Paderborn, Germany, March 1996.
- [3] T. T. Do, S. Faulkner and M. Kolp. Organizational Multi-Agent Architectures for Information Systems. in Proc. of the 5th Int. Conf. on Enterprise Information Systems (ICEIS 2003), Angers, France, April 2003.
- [4] S. Faulkner and M. Kolp. Towards an Agent Architectural Description Language for Information Systems. In Proc. of the 5th Int. Conf. on Enterprise Information Systems (ICEIS 03), Angers, France, April 2003.
- [5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In Proc. of SIGSOFT'94: Foundations of Software Engineering, New Orleans, Louisiana, USA, Dec. 1994.
- [6] JACK Intelligent Agents. <http://www.agent-software.com/>.
- [7] M. Kolp, P. Giorgini, and J. Mylopoulos. An Organizational Perspective on Multi-agent Architectures. In Proc. of the 8th Int. Workshop on Agent Theories, architectures, and languages, ATAL'01, Seattle, USA, Aug. 2001.
- [8] E. Yu. Modeling Strategic Relationships for Process Reengineering, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.
- [9] M. Wooldridge and N.R Jennings, editors. Special Issue on Intelligent Agents and Multi-Agent Systems. Applied Artificial Intelligence Journal. Vol. 9(4), 1996.