TOWARDS AN AGENT ARCHITECTURAL DESCRIPTION LANGUAGE FOR INFORMATION SYSTEMS

Stéphane Faulkner, Manuel Kolp,
IAG- School of Management, ISYS- Information Systems Research Unit, University of Louvain, 1 Place des Doyens, Belgium
Email:faulkner@isys.ucl.ac.be, kolp@isys.ucl.ac.be

Keywords: Multi-Agent Systems, Architectural Description Language, BDI Agent Model, System Architecture

Abstract: This paper identifies the foundations for an architectural description language (ADL) to specify multi-agent system architectures for information systems. We propose a set of system architectural concepts based on the BDI agent model and existing classical ADLs. We then conceptualize SKwyRL-ADL, aimed at capturing a "core" set of structural and behavioral concepts, including relationships that are fundamental in architecture description for BDI-MAS. We partially apply our ADL on a peer-to-peer document sharing example.

1 INTRODUCTION

The characteristics and expectations of new application domains for the enterprise such as e-business, knowledge management, peer-to-peer computing or web services are deeply modifying information systems engineering. Most of the systems designed for these kinds of application areas are now de facto concurrent and distributed. They tend to be open and dynamic, in that they exist in a changing organizational and operational environment where new components can be added, modified or removed at any time.

Given these new needs, we advocate the use of Multi-Agent Systems (MAS) architectures to build today's enterprise information systems. MAS architectures do allow dynamic and evolving structures and components which can change at run-time to benefit from the capabilities of new system entities or replace obsolete ones.

Such architectures become rapidly complicated due to the ever-increasing complexity of these new business IT domains and their actors: as the expectations of users and business stakeholders change day after day, as the complexity of systems, information and communication technologies and organizations continually increases in today's dynamic environments, developers are expected to produce architectures that must handle more difficult and intricate requirements that were not taken into account ten years ago, making architectural design a central engineering issue in modern enterprise information system life-cycle.

To cope with this complexity in architectural engineering, there have been, through the last decade, different proposals for providing a sound basis for describing formally system architecture and reasoning about it. In particular, a number of Architecture Description Languages (ADLs) have been proposed such as Rapide [LKA+95], Darwin [MK96], Aseop [GAO94], Unicon [SDK+95], Wright [AG94] and ACME [GMW97]. ADLs provide constructs for specifying architectural abstractions in a descriptive notation. They offer formal mechanisms for decomposing a system into architectural elements, specifying how these elements are combined to form a configuration.

Unfortunately, few research efforts aim at truly defining an architectural description language for MAS architectures. Therefore, the main contribution of this work is to propose the foundations for an architectural description language specifying multi-agent systems that are based on the *Belief-Desire-Intention* (BDI) agent model. To this end, we define, in the context of the SkwyRL project[1], an ADL specification called SkwyRL-ADL identifying architectural concepts that subsume those present in existing BDI agent-oriented development platforms such as Jack or Jade. This specification aims at capturing a "core" set of structural and behavioral abstractions, including relationships and

[1] Socio-Intentional ArChitecture for Knowledge Systems & Requirements ELicitation (www.isys.ucl.ac.be/ skwyrl)

constraints that are fundamental to the description of any BDI-MAS architecture.

The rest of the paper is organized as follows. Section 2 overviews the notions of agent and multi-agent system and identify the main concepts of the BDI model we will use in our ADL. Section 3 identifies the main features of ADLs we will use in SKwyRL-ADL. Section 4 presents SKwyRL-ADL and specifies part of it in first order logic. Section 5 briefly introduces a peer-to-peer example illustrating the use of our agent ADL. Finally, Section 6 summarizes the results and points to further work.

2 AGENT AND MAS

An *agent* defines a system entity, situated in some environment that is capable of flexible autonomous action in order to meet its design objective [WJ96].

Three key concepts support this definition:

- *Situatedness*: an agent receives input from the environment in which it operates and can perform actions, which change the environment in some way;
- *Autonomy*: an agent is able to operate without direct, continuous supervision, it has full control over its own actions;
- *Flexibility*: an agent is not only reactive but also pro-active. Reactivity means that it has perceptions of the world inside which it is acting and reacts to change in quasi real-time fashion. Proactiviness means that behavior is not exclusively reactive but it is also driven by internal goals, i.e., it may take initiative.

A *multi-agent system* can be defined as an organization composed of autonomous and proactive agents that interact with each other to achieve common or private goals [KGM01].

MAS may be either cooperative or competitive agents. In a cooperative MAS, the system has a global goal (or set of goals) and the agents that compose the MAS cooperate, possibly by performing diverse tasks, in order to achieve the global goal. This kind of systems is typically adapted to perform distributed problem solving. There is a unique high-level goal decomposed recursively into parallel activities to be performed by a set of agents.

In a competitive MAS, each of the component agents has its own set of goals that may or may not meet those of other agents. In this case the MAS is an architecture that allows agents to interact, each one to pursue personal goals and defend its own interests. This kind of systems meet typically engineering requirements of e-commerce, information retrieval applications, web services or peer-to-peer networks. In such environments, every agent generally represents either a client, who wants to obtain some resources or have some service accomplished, or a provider, who wants to sell resources or services at a certain (not necessarily financial) cost. Each agent pursues the goals of the (human or system) actor it represents, and these goals can usually be in conflict.

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. Agent models are proliferate; some include learning capabilities, others intelligent agendas based on statistics, others yet are based on genetic algorithms and so on. However, 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 [Bra87]. This approach has been intensively used to study the design rationale of agents and is proposed as a keystone model in numerous agent-oriented development environments such as Jack or Jade. The main concepts of the BDI agent model are (except 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*

In more detail, a BDI agent has a set of plans, which defines sequences of actions and steps available to achieve a certain goal or react to a specific situation.

The agent reacts to events, which are generated by modifications to its beliefs, additions of new goals, or messages arriving from the environment or from another agent. An event may trigger one or more plans; the agent commits to execute one of them, that is, it becomes intention.

Plans are executed one step at a time. A step can query or change the beliefs, performs actions on the external world, submits new goals. The operations performed by a step may generate new events that, in turn, may start new plans. A plan succeeds when all its steps have been completed; it fails when certain conditions are not met.

3 ADL CONCEPTS

Architecture description languages are formal languages that are used to specify the architecture of a system [SG96].

By *architecture*, we mean the components that compose a system, the behavioral specification for those components, and the mechanisms for interactions among them.

In order to propose an ADL for BDI-MAS in Section 4, we have identified through the literature the essential aspects of a system architecture that any ADL should be able to specify. To this end, we have examined different ADLs, surveys and attempts at identifying essential architectural characteristics and requirements [Cle96, Ver93, LV95, SG95, Tra93]. Based on this analysis, this section proposes a state-of-the-art ontology that determines a common foundation of concepts and concerns for system architecture description:

- **Components** *are units of computation or data stores. Therefore, components are loci of computation and state*.

A component in architecture may be as small as a single procedure or as large as an entire application. It may require its own data and/or execution space, or it may share it with other components.

- **Interfaces** *are set of interaction points among itself and the external world*.

All ADLs support specification of component interfaces. They differ in the terminology and the kinds of information they specify. For example, each interface point in ACME, Aseop, and Wright is called a port. In UniCon, an interface point is a player, and in Rapide a constituent. In Darwin, the interface consists of a collection of services that are either provided or required.

- **Connectors** *are used to model interactions among components and rules that govern those interactions*.

The notion of connection varies among languages. For instance, Rapide and Darwin have a very weak notion of connectors. Connections are specified with bindings between the provided service of a component and the required service of another component. In Aesop and UniCon, each connector has a collection of roles that define what participants are expected in a given interaction.

- **Configurations** (*or topologies) are connected graphs of components and connectors that describe architectural structure*.

Configurations are needed to determine whether appropriate components are connected, their interfaces match, and their combined semantics result in desired behavior. Descriptions of configurations enable assessment of concurrent and distributed aspects of architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, etc. Configurations also enable analyses for adherence to design heuristics and style constraints. In this sense, a major role of configuration is to facilitate understanding of systems at a high level of abstraction. Therefore, configuration must model structural information with simple and understandable syntax.

- **Constraints** *are properties of or assertion about a system or one of its parts, the violation of which will render the system unacceptable to one or more stakeholders*.

Constraints ensure adherence to intended component uses, enforce usage boundaries, and establish intra-component dependencies. They may be defined in a separate constraint language or using the notation of the given ADL and its underlying semantic model.

- **Hierarchical Compositions** *represent architectures as single components in other, larger architectures.*

Several ADLs provide explicit features to support hierarchical composition: ACME templates, composition elements in Darwin and UniCon, and Rapide maps. Other ADLs, such as Wright, allow hierarchical composition in principle, but provide no specific constructs to support it.

4 SKWYRL-ADL

From the generic features of the BDI agent model presented in Section 2 and the ADL concepts identified through the software architecture literature proposed in Section 3, the objective of the work has been to extract a set of concepts necessary for the definition and specification of BDI-MAS architectures. As a result, this section proposes a conceptualization of SKwyRL-ADL, our ADL devoted to such systems architectures.

Because the aim is to define an ADL, we do not consider the dynamics of the operations of programming languages that describe what the compiler should make to handle every instruction. For example, the operational semantics specifying how, to accomplish an intention in order to fulfill a given goal, an agent chooses at run-time a plan to be executed among a set of selected plans is out of the range of ADLs. In the context of this work we only define semantics relative the architectures, which comprises system components, the externally visible properties of those components, and the relationships among them.

In the rest of this section, we introduce the main elements of SkwyRL-ADL separating these key entities in two sub-models which operate at two different levels of abstraction: internal or global. The internal model captures the states of the agent and its potential behavior. The global model describes the interaction among agents that compose the MAS architecture. Due to the lack of space, we only specify some of these concepts in first order logic.

4.1 Internal Model

Figure 1 conceptualizes the main entities and relationships of the internal model. It is composed of six main design entities that are defined as follows :

Beliefs. *A belief is a finite set of objects that represents a view of the current environment of an agent.*

From a theoretical point of view, beliefs can be considered first-order logic predicates or relations. The world (i.e., the environment of an agent) consists of objects, things with individual identities and properties that distinguish them from

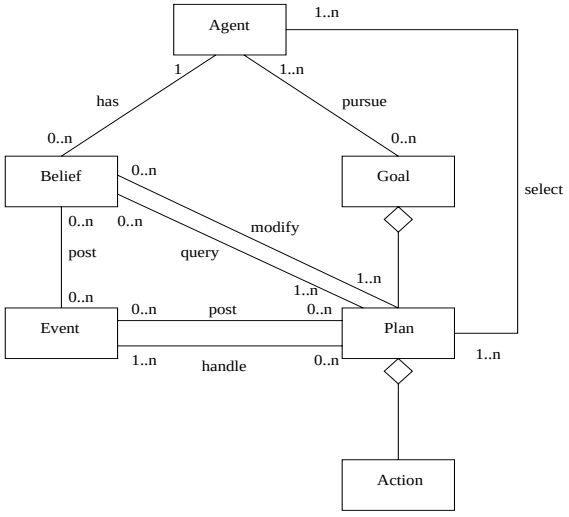

Figure 1 : Conceptual representation of the Internal Model

the other objects. Among these objects various relations hold. Some of these relations are function s(i.e., relations in which there is only one "value" for a given "input").

We specify a belief as follows:

Belief → AtomicBelief
| Belief Connective Belief
| ¬Belief

With respect to the definition, we go a step farther by specifying the beliefs also as being able to be the result of a conjunction, a disjunction or an implication of beliefs.

AtomicBelief → *Predicate*(*Term*, …)

Term → *Function*(*Term*,…)
| *Constant*
| *Variable*

A term is a logical expression that refers to an object and a predicate symbol refers to relations.

Connective → ∧ | ∨ | ⇒
Constant → *A* | X_1 | *Jhon* | ….
Variable → a | x | s |….
Function → *Mother* | *LeftLegOf* | ….

As depicted in Figure 1, beliefs are not static. Indeed, they can be modified during the execution of a plan. This modification can trigger the posting of an event, which can trigger in turn the execution of a new plan.

Goal. *A goal is a set of objects that describe an environment state that an agent wants to bring about.*

We consider goals according to four patterns [Kaos93].

- *Achieve*: $P \Rightarrow \Diamond Q$

Property Q holds in current or some future state

- *Cease*: $P \Rightarrow \Diamond \neg Q$
- *Maintain*: $P \Rightarrow £ Q$

Property Q holds in current and all future states

- *Avoid*: $P \Rightarrow £ \neg Q$

With respect to beliefs, goals can be specified as follows:

Goal → *achieve*(Belief)
| *cease*(Belief)
| *maintain*(Belief)
| *avoid*(Belief)

These goal patterns influence the set of possible agent behaviors: *achieve* and *cease* goals generate actions, plan, or events, while *maintain* and *avoid* goals restrict them.

When a goal is required, the agent identifies a set of plans to achieve or maintain this goal. From then on, the agent chooses according to its current beliefs which of these plans will be executed.

Action. *An action is a basic executable command of agent behaviour in order to achieve goals or accomplish tasks.*

The type of action that agents can perform may be classified as either external (the domain of the action is the environment outside the agent) or internal (the domain of the action is the agent itself). We will explain in Section 4.2 that external actions are specified as provided or required services and why they play an important role in the definition of the system topology.

Plan. *A plan defines the sequence of actions to be chosen by the agent to accomplish a task or achieve a goal.*

Plans are selected by agents. Selected plans constrain the agent's behavior and act as intention. As said in Section 2, intentions represent the deliberative states of the agent, i.e., which plans the agent has chosen for possible execution. Run-time specification of these deliberative states should be taken into account when defining the operational semantics of the system. At the architectural level we only consider in the ADL the basic components needed to define intentions.

A plan consists then 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 to be executed or an goal to be satisfied;
- and finally a set of internal actions that specify what happens when a plan fails or succeeds.

Event. *An event is something that happens in the system that can be perceived, and it is either a goal (a new goal or the removel of a goal), a belief (a new belief or the removel of a belief) or a plan (the success or failure of a plan).*

We specify an event as follows:

Event → TriggerEvent
| *Succeed*(Plan)
| *Failed*(Plan)

TriggerEvent → TriggerFunction X Trigger
TriggerFunction → Add | Rem
Trigger → Belief | Goal

MAS are event-driven in the sense that agents start interacting by initiating and perceiving events. Events are the origin of all activity within the system. In the absence of event an agent sits idle. Whenever an event occurs, an agent initiates either a plan or a set of plans to response to that event. In this last case, the agent chooses between the plans it has available to achieve its goal.

We defined two types of events:

- *Internal event*, that an agent posts to itself;
- *External event*, that an agent sends to other agent or to its environment;

According to the definition of event, both types are specified considering the nature of the event, which can be a goal, a belief or a plan. The key difference between belief, plan or goal events is how an agent selects plans for execution. For belief

and plan events, the agent selects the first applicable plan for that event and executes an instance of that plan only. The handling of goal event is more complex. An agent can assemble a set of plan for the goal event and apply a sophistical heuristics to choose the appropriate plans. However, for this matter, at the architectural design level where ADLs are defined, we remain completely independent from such heuristics, considering that they depend directly on the used programming environment.

4.3 Global Model

The global model describes the interaction among agents that compose the MAS. Figure 2 illustrates the main entities and relationships the model captures. The design entities are defined as follows:

Interface. *An interface defines a collection of "connection points" through which agent interacts with its environment.*

An interface is a specification of how an agent "appears" to the rest of the system. Its primary constituents are a set of effectors and a set of sensors which model the points through which an agent interact. An effector provides one or several services for other agents and/or human users. Then, a sensor requires one or several services from other agents and/or human users. Thus, for each interaction, there is always a correspondence between a service provided by a effector and a service required by a sensor. Other kinds of interaction may have more than two services: an event broadcast interaction might have a single event-announcer service and an arbitrary number of event-receiver services. We will see also that this correspondence plays a role in the configuration specification of the system.

Service. *A service is an action or a sequence of actions (plan) that involves an interaction between an agent and one or several agents or human users.*

We distinguish two natures of services: *Provide* and *Require*. Provided services are services that agents provide to other agents or human users of the system. Required services are services that agents require from other agents or human users of the system.

Configuration. *A configuration is an interconnected set of agent instances.*

A MAS architecture is represented as a configuration of instantiated agents components. The topology of the system is defined by a set of bindings between services provided by effector instances and services required by sensors instances. The configuration separates the descriptions of composite structures from the elements in those compositions. This allows reasoning about the composition as a whole and changing the composition without having to examine each of the individual components in a system.

Architecture. *An architecture models the full set of design information defined within an architectural specification.*

The architecture concept models a complete system specification. Each design has at least one configuration corresponding to the top-level system model. However, our ADL also supports hierarchical system descriptions; that is, an agent may be further specified as being designed by a configuration. Architecture maintains the set of all of the configurations that have been defined.

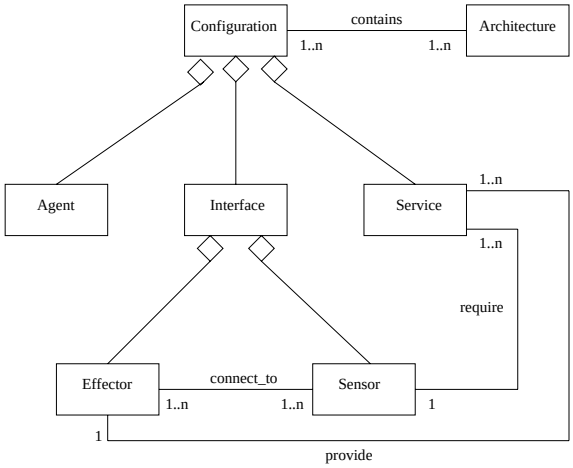

Figure 2 : Conceptual representation of the Global Model

5 A PEER-TO-PEER SYTEM

In order to illustrate our ADL this section presents a peer-to-peer prototype system we are developing, called SciDOC-*MEDIA*, aimed to share scientific documents among several university communities. We describe partially this architecture with a set of examples applying architectural concepts we have proposed in Section 4.

SciDOC-MEDIA is a peer-to-peer (P2P) network in which each party has a same communicational capability and can initiate a communication session. To share documents on the *SciDOC-MEDIA* network, a user (node A for example) starts with a networked computer that runs one of the *SciDOC* clients. Since this node works both as a server and a client, it is referred to as a (SciDOC) "servent" (both a SERVer and a cliENT). Node A can query the contents of the document shared by another SciDOC-enabled networked computer (node B for example). Node B in turn can query all neighboring nodes (nodes C,D,E and F for example) for A. This behavioral pattern continues recursively with each new level of nodes. Figure 3 illustrates this model.

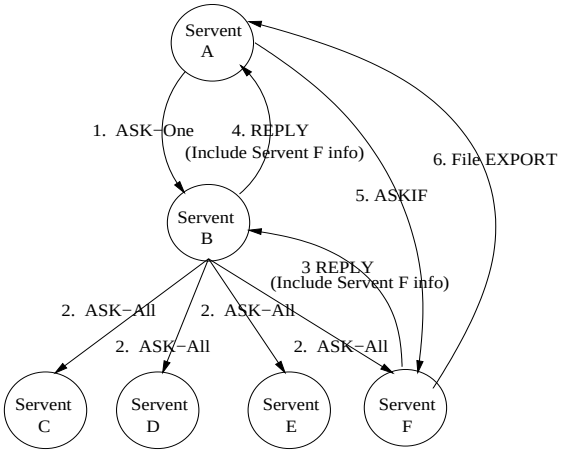

Figure 3 : The SciDOC Decentralized P2P Model

Part of the belief representation of a SciDOC SERVENT consists of a predicate *document* representing a document shared by a SERVENT in the network. It has four arguments:

Belief*: document*(*Author, Title, Editor, Year*)

Localisation(*doc*, *Disk, Path*) is a predicate that indicates that a document *doc* is stored locally on the disk *Disk* and is accessible from the path *Path*.

It is assumed that each SERVENT can pursue a goal that consists in acquiring new documents. Such goal can be defined with the goal pattern *Achieve*:

Goal: *achieve* (*document(autor?, title? editor?, year?)*
∧ *localisation(doc, disk!, path!*)
∧ *disk* ≠ LOCAL ∧ *path* ≠ LOCAL)

This goal defines a desirable state made of the conjunction of four beliefs. It represents a document identified by an author, a title, an editor, and a publication year and has a localisation on the SERVENT. Note that question marks indicate inputs, and exclamation marks indicate outputs.

With respect to the above belief and goal specification and the scenario of Figure 3, a plan can be defined for the SERVENT A as follows:

Plan:
Invocation condition
achieve (*document(author?, title? editor?, year?)*
∧ *localisation(doc, disk!, path!*)
∧ *disk* ≠ LOCAL ∧ *path* ≠ LOCAL)
Context
document(autor?, title? editor?, year?)
∧ *l ocalisation(doc, Null, Null*)
Body
ASK-ONE(KB_b,(*document(autor?,title?,editor?,year?)*
Failed
ASK-ONE(KB_i,(*document(autor?,title?editor?, year?)*)
with i ≠b
Fail Condition
(TTL > 100 ∧ (REPLY ())
Succeed
ASKIF(Ar, EXPORT(doc.file))

The goal consisting in acquiring new documents serves as the invocation condition (i.e., trigger event) for the above defined plan. The context contains a precondition that ensures that the required document is not stored locally on the SERVENT disk. The body specifies only one action ASK-ONE[2] that allows to know if a SERVENT B has the required document in its knowledge base KBb. The plan fails when the failed condition is true (i.e., the Time-To-Live (TTL) packets linked to the ASK-ONE action exceeds 100 micro seconds and there was no response from the network). Then, the SERVENT A re-executes an ASK-ONE for other SERVENTs. When the fails condition is false, the SERVENT A executes an ASKIF action that asks the SERVENT storing the required document to export the document file.

The following specification illustrates a configuration. Here we make two assumptions due to the lack of space: (i) the configuration describes only the scenario illustrated in Figure 3; (ii) SERVENT interfaces only contain a single type of effector and a single type of sensor.

Configuration SciDOC
Agent SERVENT
Effector StandardEF
Sensor StandardSN
Instances
A_i, A_j, $A_{xk[k= 1..n]}$: SERVENT
ef_i, ef_j, $ef_{xk[k= 1..n]}$: StandardEF

[2] The ASK-ONE semantics, as well as the other actions semantics used to specify the plan, are defined from the Speech Act theory.

sn_i, sn_j, $sn_{xk[k=1..n]}$: StandardSN

with ef_i, $sn_i \in$ Interface(A_i);
ef_j, $sn_j \in$ Interface(A_j);
ef_{xk}, $sn_{xk} \in$ Interface(A_{xk});

Bindings

$A_i.ef_i$.ASK-ONE To $A_j.sn_j$
$A_j.ef_j$.ASK-ALL To $A_{xk}.sn_{xk}$
$A_{xt}.ef_{xt}$.REPLY To $A_j.sn_j$ ($\exists$ t, t=1..n)
$A_j.ef_j$.REPLY To $A_i.Sn_i$
$A_i.ef_i$.ASKIF To $A_{xt}.Sn_{xt}$
$A_{xt}.ef_{xt}$.EXPORT To $A_i.Sn_i$

End SciDOC

6 CONCLUSION

This paper has advocated the use of Multi-Agent System architectures to design today's information systems that must now be based on open architectures to accommodate new components and meet new requirements. Multi-Agent Systems (MAS) do actually provide for open and evolving architecture which can change at run-time to exploit the services of new agents, or replace existing ones. However, few efforts have been made to define an architectural description language (ADL) for this type of architectures. Based on the analysis of existing classical ADLs and the BDI agent model, the paper has identified a set of system architectural concepts to propose an ADL for BDI-MAS architectures.

The research reported here calls for further work. We are currently working on :

- the clarification of the relationship between the agent approach to architectural design and that of more traditional (e.g., object) ones;
- the identification of a suitable set of core abstractions, inspired by an organizational metaphor, to be used during the design of multi-agent systems architecture;
- the development of a clear methodology, centered around these organizational abstractions, for the design of multiagent systems architectures.

REFERENCES

[AG94] R. Allen and G. Garlan. *Formal Connectors*. Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994.

[Bra87] M . E. Bratman. Intention, Plans and Practical Reason. *Harvard University Press*, 1987.

[Cle96] 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.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[GMW97] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language, In *Proc. of CASCON 97*, Toronto, pp. 169-183 Jan. 1997.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. on Software Engineering*, pp. 336-355, April 1995.

[LV95] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.

[KGM01] 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.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of the 4th Symposium on the Foundations of Software Engineering* (FSE4), pp. 3-14, San Francisco, CA, October 1996.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, pp. 314-335, Apr. 1995.

[SG94] M. Shaw and D. Garlan. *Characteristics of Higher_Level Languages for Software Architecture*, Thechnical Report, CMU-CS-94-210, Carnegie Mellon Univ., Dec. 1994.

[SG95] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. vanLeeuwen, editor, *Computer Science Today*, pages 307–323. Springer-Verlag, Berlin, 1995.

[Tra93] W. Tracz. LILEANNA: A Parameterized Programming Language. In *Proc. of the 2nd Int. Workshop on Software Reuse*, pages 66-78, Lucca, Italy, 1993.

[Ver93] S. Vestal. *A Cursory Overview and Comparison of Four Architecture Description Languages*. Technical Report, Honeywell Technology Center, February 1993.

[WJ96] M. Wooldridge and N.R Jennings, editors. Special Issue on Intelligent Agents and Multi-Agent Systems. *Applied Artificial Intelligence Journal*. Vol. 9(4), 1996.