A Multi-agent Perspective on Data Integration Architectural Design

Stéphane Faulkner, Manuel Kolp, Tai Nguyen, and Adrien Coyette

Information Systems Research Unit, University of Louvain,
1 Place des Doyens, 1348 Louvain-la-Neuve, Belgium
{faulkner, kolp, nguyen, coyette}@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 data 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 data integration system design as illustrated in this paper.

1 Introduction

Architectures for integrating data 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 data sources to obtain an integrated answer. The sources may be conventional databases or other types of data, such as collections of Web pages.

Designing data 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 data 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 data 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, a number of architectural description languages (ADL) [1] and architectural styles [4] 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

M.Gh. Negoita et al. (Eds.): KES 2004, LNAI 3213, pp. 1150–1156, 2004.
© Springer-Verlag Berlin Heidelberg 2004

this gap, we have defined, in the SKwyRL[1] project, architectural styles for multi-agent systems based on an organizational perspective [2] and have proposed in [3] 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 data 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 describes the design of the global architecture with an organizational style. Section 3 presents part of the formal specification with SKwyRL-ADL. Section 4 concludes the research.

2 Organizational Architecture for Data Integration

Figure 1 models the application architecture using the i* model [6] following the joint-venture organizational style. 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…).

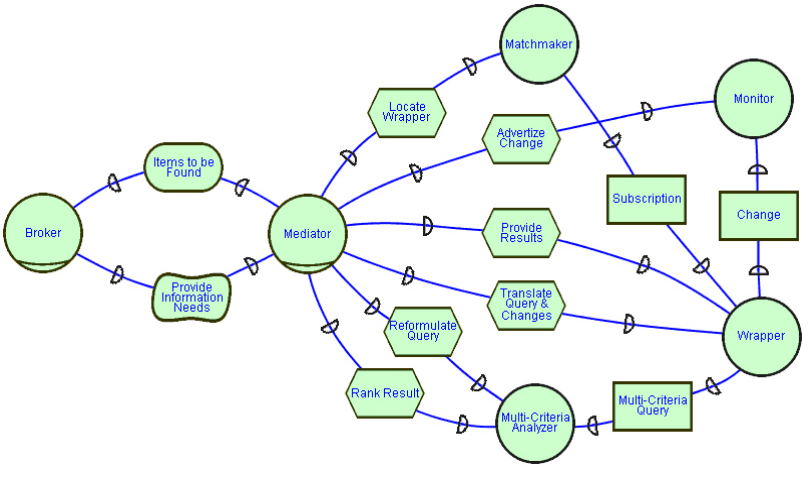

Fig. 1. The application architecture in joint-venture

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

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

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. *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 *depender* and the *dependee*. The *depender* 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* 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 1, actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; dependencies have the form *depender* → *dependum* → *dependee*.

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 data 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 agent are connected to each data 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 data 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 Formal Architectural Specification

The architecture described in Figure 1 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. 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 2 shows a partial 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 behaviours.

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 2 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)]
    …
    Effector[provide(items_found)]
    …
KnowledgeBase :
    DataManagement_KB
    …
Capabilities :
    Handle_Request_CP
    …}
```

Fig. 2. 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.

The required query translation service is described in greater detail in figure 3. 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.

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

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

Fig. 3. A Service Specification

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 4 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 }
```

Fig. 4. A Knowledge Base Specification

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.

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

Fig. 5. A Capability Specification

Figure 5 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 tow plans: *DecompNmlRq* is used to decompose a normal request, *DecompMCRq* to decompose a multi-criteria request.

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*.

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.

```
Configuration GOSIS
Agent  Broker[nb: 1…]; Mediator[nm: 1…];Wrapper[nw: 1…nS];
       Monitor[nmo: 1…nS]; Matchmaker; Multi-Critria-analyzer
Service   Tell(query_translation); sk(query_translation);
          Achieve(result) ; 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
       ME nm.Askquerytrans  --- Tellquerytrans.WRnw;
       ME nm.Achres  --- Tellres.WRnw;
       ME nm .Asksubs --- Tellsubs.MA;
….
End GOSIS
```

Fig. 6. GOSIS Parameterized Configuration

Part of the configuration with instance declarations and collaborations is given in Figure 6 "(*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.

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 data integration system engineering. The paper has proposed a validation of the approach: it has been applied to develop GOSIS, an data integration platform implemented on the JACK [5] agent development environment.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] JACK Intelligent Agents. http://www.agent-software.com/.

[6] E. Yu. Modeling Strategic Relationships for Process Reengineering, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.