Facilitating the Specification Capture and Transformation Process in the Development of Multi-Agent Systems

Aluízio Haendchen Filho[1], Nuno Caminada[2], Edward Hermann Haeusler[1], Arndt von Staa[1]

PUC-Rio[1] – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, Rua Marquês de São Vicente 225, CEP 22453-900, Rio de Janeiro, RJ, Brasil
{aluizio,hermann,arndt}@inf.puc-rio.br
UniverCidade[2] – Centro Universitário da Cidade do Rio de Janeiro (UniverCidade/NUPAC)
caminada@atividade.com.br

Abstract. To support the development of flexible and reusable MAS, we have built a framework designated MAS-CF. MAS-CF is a *component framework* that implements a layered architecture based on contextual composition. Interaction rules, controlled by architecture mechanisms, ensure very low coupling, making possible the sharing of distributed services in a transparent, dynamic and independent way. These properties propitiate large-scale reuse, since organizational abstractions can be reused and propagated to all instances created from a framework. The objective is to reduce complexity and development time of multi-agent systems through the reuse of generic organizational abstractions.

1 Introduction

The characteristics and expectations of new application domains surrounding distributed systems have lead to the development of dynamic and evolving structures. After the advent of the Internet and with the recent emergence of new technologies, the application domain of MASs is expanding and nowadays it is used in many areas, such as e-business, web-services, knowledge management and now enterprise information systems [Faulkner2001, Griss2003, Adam2004, Giorgini2004]. Agent technology represent an extraordinary opportunity for information systems and corporate applications, because agents must be capable of managing and organizing information, recognizing personal tastes and making increasingly important decisions on behalf of their owners.

Nevertheless, the development of multi-agent systems is not trivial. To avoid the task of designing each new system, we need tools to help in the MAS construction, and by extension it is desirable to also have tools for reusing previous designed architectures and their relationships. There is a considerable research effort towards the development of frameworks for agent-based systems [Sycara1999, Wooldridge2000, Evans2001, Bellifemine2001]. Each framework has different

application specific particularities, such as social capabilities, reasoning, flexibility for dynamic compositions, interoperability and so on.

Most approaches, however, focus on the reuse of application-specific concepts at the analysis, design and implementation levels (roles, protocols, agent architectures). Little research is conducted towards generic (i.e, application-independent) models [Faulkner2001, Zambonelli2002, Holvoet2003, Griss2003]. There is a large potential of reusing generic "organizational abstractions" – such as structures and patterns – for generic (i.e, application-independent) models [Zambonelli2002]. Reuse of generic software is recognized within the object-oriented community and has lead to the concepts such as design patterns and frameworks [Pree1999, Fayad1999].

The main focus of our work is the reuse of abstractional organizations applied to the development of multi-agent systems. Reuse an abstract architecture allow us not only to reuse the design and the implementation of the architectural software, but also the reuse of important individual agent properties, such as interaction, adaptation and collaboration, which can be completely or partially resolved at the architectural level. On the other hand, by freeing the developer from the task of implementing these complex properties on the agent, the work becomes simpler and can be better focused on the maintenance of the knowledge structure and on the learning capabilities of the agent.

This paper is structured as follows: the next section briefly describes the state-of-the art regarding agents and multi-agent systems. Section 3 describes the abstract architectural model, the communication model and interface specification. Section 4 describes the interaction model, formalized by means of service ontology. Section 5 describes how the architecture behavior has been formalized and how the specifications are being stored and transformed into reliable code. Related works are discussed in Section 6 and Contributions are listed in Section 7.

2 Agent and Multi-Agent Systems

We have examined and identified through the literature the essential aspects surrounding agent-based technology. This section briefly presents some important concepts that will be used on the course of this work, namely agents and multi-agent systems.

2.1 Agents

There is no universally accepted definition of the term agent. Part of the difficulty to define agent arise from the fact that for different domains of applications, the properties associated with the agent concept assumes different levels of importance. There are many types of software agents with different characteristics such as mobility, autonomy, collaboration, persistence and intelligence.

The behavior of an agent depends on, and is affected by, the incorporated agency properties: interaction, adaptation, autonomy, learning, mobility and collaboration. Such properties were based on previous studies [Kendall1999, OMG2000, Garcia2001]. We have use the properties as follows, based on [Garcia2001]:

- *Interaction*: an agent communicates with the environment and other agents by means of sensors and effectors. These are available via the agent's provided and required interfaces;
- *Adaptation*: an agent should adapt its state and behavior according to new environmental conditions;
- *Autonomy*: an agent has its own control thread and can accept or refuse a request; in other words, by autonomy we understand the capacity of the agent to execute its activities without human intervention;
- *Learning*: an agent can learn on previous experience while interacting with its environment;
- *Mobility*: an agent is able to transport itself from one environment to another to achieve its goals;
- *Collaboration*: an agent can cooperate with other agents in order to achieve its goals and the system goals.

According OMG [OMG2000], *autonomy*, *interaction* and *adaptation* can be considered as fundamental properties of software agents, while learning, mobility and collaboration are neither a necessary nor sufficient condition for agenthood. There are several types of software agents, including information agents, user agents, interface agents and mobile agents. Each agent type has different application specific capabilities and agency properties. In order to have autonomy, an agent must possess a certain degree of intelligence allowing it to survive in a dynamic and heterogeneous environment [Correa1994]. Therefore, there is general consensus that autonomy is one of the central properties to the notion of agent.

2.2 Multi-Agent Systems

There are several different ways to organize multi agent systems. In any given case, the best way depends on the purpose and objectives of the system, thus there are several types of multi-agent systems, each with its own particularities such as social capabilities, reasoning, interoperability and so on. Jennings [Jennings1996] proposes a framework that provides a structure to analyze and classify the activities of multi-agent systems according to two different perspectives: (i) the agent perspective: focuses on the characteristics of the agent involved with the MAS, such as internal architecture, structure and maintenance of knowledge, and abilities of reasoning and learning; (ii) the group perspective: includes group aspects such as organization, coordination, interaction and negotiation.

In MESSAGE [Evans2000], MAS architecture is defined through an organizational model, focused on the structure of the organization and the relationship between the agents it contains. The organizational model also describes mechanisms for conflict resolution and rules that enable agent groups to function as a unit serving a common purpose. Agents are identified based on a goal-oriented model, where organizational goals are decomposed and associated with tasks. Goal decomposition is carried out recursively, until the tasks associated with the goal can be completely fulfilled by an isolated agent or in collaboration with other agents. Agents are connected by organizational relationships (such as superior-subordinate and client-provider), proceedings of control management, workflows and interactions. Internal architecture and maintenance of the knowledge structure applies an approach similar to BDI (Beliefs, Desires, Intentions).

On the design of interoperable agents, JADE [Bellifemine2001] is a framework focused on interoperability based on the standardization of the language of knowledge. JADE can be considered an agent middleware that implements a platform and a development framework. The interaction model is implemented according to FIPA [FIPA2000] protocols. FIPA provides a standard language of communication based on protocols, an ontology necessary for the interaction between the agents from the system and from other systems. JADE provides an API to organize the system starting with a set of generic system services and agents. Services are transported through an interface mechanism to send/receive messages to/from other agents.

RETSINA [Sycara1999] focuses the agent architecture in a software infrastructure that allows heterogeneous agents to interact on the Internet. The RETSINA framework provides an abstract basic agent architecture consisting of, and integrating with, reusable modules and each module of an agent operates asynchronously. The RETSINA definition of multi-agent systems is driven by the vision that heterogeneous agents that autonomously organize their own social structures should populate multi-agent societies.

The descriptions show different ways to organize MAS. Nevertheless, most approaches focus the reuse in specific application concepts and on the individual properties of the agent, such as protocols, roles and internal architecture. Little research on the domain of multi-agent systems has been conducted emphasizing the reuse of generic organizational abstractions [Faulkner2001, Zambonelli2002, Holvoet2003, Griss2003].

3 The Architectural Model

In this section we present the main models that compose the framework architecture, thus, the abstract model, the structural model, the interface model and the logic model are described and commented.

3.1 The Abstract Model

The architecture of a multi-agent system can naturally be viewed as an organized computational society of individuals. For this reason, organizational abstractions should play a central role in the analysis and design of such systems. Zambonelli and Wooldridge [Zambonelli2002] state that "the introduction of high-level organizational abstractions can lead to cleaner and more manageable and reusable MAS design." Also according to Zambonelli, the organizational abstractions facilitate the design process because it leads to a cleaner separation between the component level (i.e., intra-agent) and system-level (i.e., intra-system). Holvoet [Holvoet2003] argue that "programming in the large" for reactive MASs should imply a reuse method that allows two things: (i) to describe MASs in an abstract, application-independent way and (ii) to reuse such abstract multi-agent system through application-specific adoptions.

In order to address these necessities, a few basic requisites of the model must be introduced. First we define MAS from an organizational view as a set of autonomous agents (possibly pre-existent) which common objective is the solution of a given problem [Jennings1996]. Nevertheless, the designer does not have to be focused on the solution of a specific problem. New problems may arise in the context of the MAS, and the society must be able to solve these new problems in collaboration. This can be achieved through the inclusion of new agents building compositions with pre-existing agents or by replacing obsolete agents. Therefore, the abstract model must provide an architecture that facilitates the inclusion of new agents at any given moment as new problems arise.

During the analysis phase, an understanding of the system and its structure can be done. In our case, this understood is captured in the system's organization, via architectural model. We view a organization as a collection of agents that provide and perform services, and take part in systematic, institutionalized patterns of interactions with other agents regulated by the architecture. Departing from the goals of the organization, services can be identified and allocated to new agents or to pre-existing ones.

3.2 Proposed Architecture

Our architecture was designed supported by the basic concepts present in component frameworks [Szyperski2002]. A component framework is a set of interfaces and interaction rules that govern how components "plugged into" the framework may interact. In particular, a component framework forms a framework that composes instances not based on directly declared connections or derivations (such as inheritance of a class framework), but based on the creation of contexts and the placement of instances in appropriate contexts [Szyperski2002]. Beyond the similar names, almost identical visions and superficially similar construction principles, component frameworks are very different from class frameworks [Bosch1999,

Fayad1999] since the inheritance implementation is not commonly used between a component framework and the interfaces it supports.

Figure 2 illustrates the two main parts that compose our structural model: *System* and *Infrastructure*. *System* defines a structural model for the domain-specific MASs. We define domain according to [Sodhi2000, Tracz1994] as the space of the problem for a family of applications with similar requirements. *Infrastructure* defines a part that contains components that provide generic services, such as database access, translation services, HTTP services, GUI builders and others.

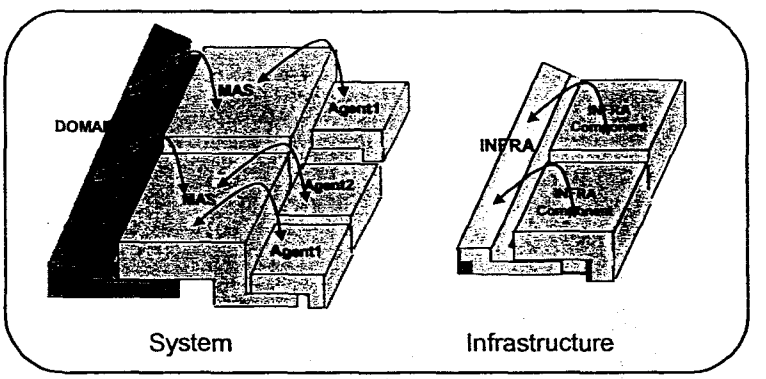

Figure 2 – The MAS-CF generic architecture

System can be seen in the left side of the Figure 2. It defines a three-tier architecture composed by the elements *Domain*, *MAS* and *Agent*. *Domain* is a component system, *MAS* is a component framework, and *Agent* is an abstract model for the instances plugged on the MAS. The Domain tier implements a set of rules of interaction that allows the communication and the sharing of services between different MAS and allows the communication between systems located in different domains. Different MAS located in a given domain can be plugged on the tier Domain. Note that tiers are described side by side with each other, while layers sit on top of each other. Traditional class framework merely structure individual components, independent of the placement in a tiered architecture. In the same way that MASs can be plugged on the Domain tier, agents can be plugged on the MAS tier.

Represented on the right side of the Figure 2, *Infrastructure* is a two-tier architecture where the Infra is a component framework and the generic Infra Components are instances of the Infra component framework. The communication between the System and Infrastructure is supported by an ontology, which describes the services and how they can be accessed. Details will be shown in the Section 4.

3.3 Communication Model

Based on fundamental principles present in component frameworks, we have defined the communication model considering that the exchange of information between agents will be implemented as connections between agents and the architecture. The objective is to allow the sharing and distribution of services in a transparent, independent and autonomous way. An agent or component is visible to the architecture and can communicate generating events, which trigger connections rules in the architecture. The communication is indirect, via a component framework that mediates and regulates component interactions. Figure 3 shows the communication model on the proposed architecture.

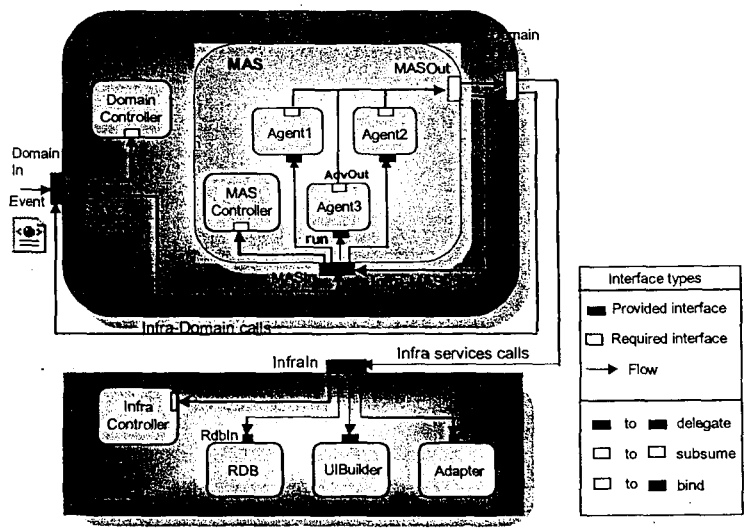

Figure 3 - The communication model

We use similar notation to SOFA [Plasil2002] to describe the communication between interfaces. Three different types of connections are distinguished: (i) *delegate*: a connection between a provided interface of a component and a provided interface of a subcomponent; (ii) *subsume*: a connection between a required interface of a subcomponent and a required interface of a component and (iii) *bind*: a connection between a required-interface and a provided-interface between two subcomponents. We have considered that the information flow between connections in bi-directional. The Java Virtual Machine places call returns in a stack. After the execution of an event, the system returns to the caller.

Services requests arrive from the environment through the interface *DomainIn*. These requests are decoded by the *DomainController* — which acts as an *abstract factory* [Gamma1995] — and are sent by the service to the responsible agent. Just as

the *DomainController*, *MASController* and *InfraController* work as *abstract factories*. They encapsulate knowledge about which concrete classes are used for the system, and conceal the way that the instances of these classes are created and joined. It permits the configuration of the system with agents "product" that can vary widely in structure and functionality. As seen in the previous subsection, the concept of component framework can be applied in such a way that component frameworks are themselves components "plugged" into higher-tier component frameworks. Thus, by construction, a component framework accepts the insertion of instances at run-time. Agents and Infra Components can be dynamically registered and plugged on the framework.

3.4 Interface Model

One of the main ideas underlying frameworks is that semi finished components can be represented by abstract classes. Their purpose is to standardize the *class interface* for all instances or subclasses. Subclasses and instances can only augment the interface, and not change the names and parameters of methods defined in a superclass [Pree1999]. The term contract [Pree1999, Szyperski2002] is used for this standardization property: instances of subclasses of a class *A* support the same contract as supported by instances of *A*. A contract is a specification attached to an interface that mutually binds the client and the providers (implementers) of that interface. Thus, the semi-finished or ready-to-use components and agents of our framework can be implemented based on the contract of the abstract class.

On the lowest level tiers, the abstract class *Agent* provides two interfaces: a provided interface designated *AgentIn* and a required interface designated *AgentOut*. *AgentIn* provides a channel of communication through which agents can absorb events and is a flexible hot-spot [Pree1999]. The *AgentOut* interface establishes a communication channel from where services from other systems, agents or components may be requested. To this end, it is only necessary to agree to the contract established by the interface. The *AgentOut* interface is a frozen-spot. Note that *Agent* here represents a generic term. In practice, the interface assumes as prefix the name of the agent and as suffix the expressions *In* and *Out*. The two interfaces are encapsulated into the semi-finished abstract class *Agent* when instanced through the framework. The basic syntax of the contract is as follows:

```
public void AgentIn(String service, Vector in, Vector out)   → sensors
public void AgentOut(String service, Vector in, Vector out)  → effectors
```

The parameter *service* (String) defines the name of the requested service. The parameters possess semantic meaning similar to IDL CORBA. They can be of type *in* (flow from client to object) or *out* (flow from object to client). The operation result, whenever there is one, is essentially a distinguished *out* parameter. The specification of highly structured messages introduces a level of complexity, since the parameters frequently represent complex types or data structures, such as vectors of objects. The

type Vector used on the *in* and *out* parameters make possible to use heterogeneous types of fields, such as Objects, arrays, Strings, and so on.

For the components of the Infra tier, only the provided-interface is instanced. Contrary to agents, components do not communicate among each other. As independent processing units, they do not request external services from other components or agents.

3.5 Logical Model

The UML provides the package mechanism [Larman1997] for the purpose of illustrating groups of elements or subsystems. Such a diagram may be called an architecture package design. A package defines a nested name space, so elements with the same name may be duplicated within different packages. Graphically, a package is shown as a tabbed folder; subordinate packages or classes may be within it. Figure 4 illustrates a more detailed breakdown of common packages in the architecture of the framework.

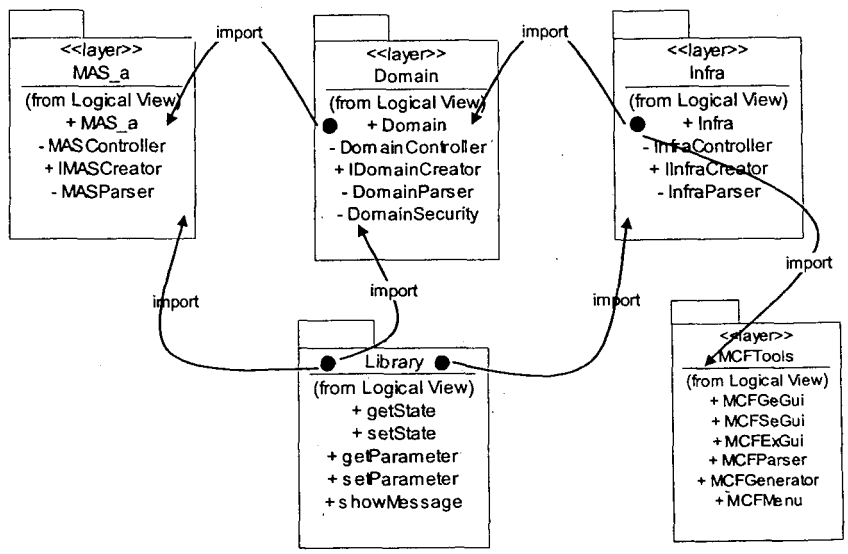

Figure 4 – Architectural units expressed in terms of UML packages

The framework contains a set of five packages: Domain, MAS, Infra, Library and MCFTools. Inside each package the encapsulated classes are listed. The three packages shown on the top represent the main tiers of the framework: Domain, MAS and Infra. Note that the three packages contain classes with the suffixes *Controller*, *Creator* and *Parser*. As seen on previous sections, the classes sporting the suffix

Controller represent *abstract factories*, responsible for the dynamic creation of instances. The *Creator* interfaces (starting with the letter *I*) define a standard signature for the instances that can be created dynamically, establishing a plug-and-play structure. The classes sporting the *Parser* suffix implement programs that parse service catalogs (detailed in the next section) to retrieve the specification of the agent or component responsible for the execution of the service. When the agent is retrieved, it is delivered in the form of a *String* from the *Parser* class to the *Controller* class, which implements a *factory method* [Gamma1995] for the dynamic creation of instances.

The two packages shown bellow on Figure 4, *Library* and *MCFTools,* supply generic support services to the main packages of the framework. Library contains some classes that supply important generic services to the programs that control the interaction flux and the synchronism between processes. The classes *setState* and *getState* are responsible for the synchronism between processes. Class *setState* (producer) stores in a hashtable the next state for the action to be executed during the transition. The data is indexed based on a ID created for each instance, and associated to the state and corresponding action. Class *getState* (consumer) whenever called upon, retrieves the state stored in the hashtable and delivers to the process the instance and the action to be executed.

The MCFTools package provides a public interface to support the tasks of instancing the architecture and the elements, along with the necessary support for the specification of the service catalog. To this end, it makes a set of GUI classes available, such as *MCFMenu, MCFGeGui, MCFSeGui. MCFMenu* is the class that provides a common interface to a group of other components of the package and system, implementing a pattern *facade* [Gamma1995, Larman1997]. The disparate elements may be the classes in a package, a framework or a subsystem (local or remote). Along with the GUI classes, the package maintains a class called *MCFParser* that captures (when the architectural elements are instanced) the specifications described by the GUIs and stores it in the XML file. Finally, the *MCFGenerator* class is responsible for code generation, working inside the standards established by the standard code structure used by the framework (as per Section 5.2)

4 Interoperability

Consider the high level component Infra. New components, which implement generic services, can be plugged at run time; new services must be available to agents at run time. How to make new services available to the agents? How to allow agents to interact with each other without knowing in advance which services are available? The representations of the architecture were not sufficient to serve as a listing of all services provided. When a new agent is registered or instantiated by the framework, its services are registered in a XML ontology in the form of a services catalog.

The use of ontology serves us as a formal specification of the catalog of services provided. Every agent/component operating within the System or Infra part must abide to the specifications dictated by the services ontology. The same is true for components. Figure 5 shows how services registered on the catalog may be accessed through the controller components present on the layers. Different components access specific sections of the catalog and obtain information such as component instances, location of services and descriptions of the communication protocols.

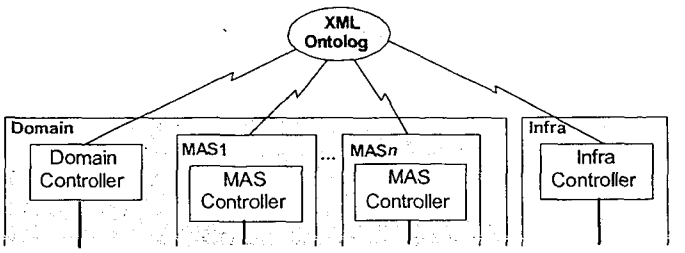

Figure 5 - Relationship between components and XML ontology

List 1 shows an example of how a services catalog can be structured in the form of an ontology. The tags *name* and *description* supply basic information about services provided by agents or by components. The *initiator* tag indicated the agent responsible for the execution of the service and the path tag indicates the physical location of the agent. It may be a physical address or a URL. The *type* tag indicates the type of protocol being used by the agent to deliver the message, initiate a conversation or supply a service.

```
- <Services>
  - <service>
      <name>Advising Receive</name>
      <description>...</description>
      <initiator>Advisor</initiator>
      <type>MAS-CF</type>
      <path>D\AcademicApplication\Advisement</path>
      <domain>Academic Applications</domain>
      <mas>Electronic Advisement</mas>
      <message>Contract MAS-CF</message>
    </service>
  </Services>
```

List 1 - XML specification of the catalog of services

The *Initiator* is the agent responsible for starting the execution of the service. The *Type* indicates the type of protocol used to deliver the message and to supply a speech act or a service. In this case, all tags are automatically retrieved from the specification and stored in XML format. Also present are the *name* and *description* tags, which supply basic information about the service. The XML catalog is critical to the system and during use a working copy is made to ensure system reliability. If the working

copy fails a new copy is reconstituted from the original. Besides, the information contained on the XML catalog can be reconstituted from the interfaces on the original XML system specification through the use of special tools.

Semantic heterogeneity is one of the chief focus of any multi-agent system, this heterogeneity expresses the issue that any two interoperating agents must be certain when using a vocabulary of terms, or translations thereof, that they are using the same concepts with the same relevant inferences of relations as the other communicating agent [Sycara2003]. Two heterogeneous interoperating agents must be certain when using a vocabulary of terms or translations (FIPA to MAS-MF, for example) that they are using the same concepts with the same relevant inferences of relations as the other communicating agent. We argue that ontology, commonly defined in the literature as a *specification of a conceptualization*, is the representation that will provide this requirement [Gruber1998].

A conceptualization can be concretely implemented, for example, in a software component. Different types of ACL (Agent Communication Language) can be identified via *Type* tag and services are provided by adapter components to translate the MAS-CF messages to/from KQML [Finin1997], FIPA, UCL [Montesco2001] and other ACLs. It decodes the calls that arrive from the environment and identifies the language spoken by the agent, for example KQML or FIPA. These components can be registered and plugged into the Infra tier.

5 Describing and Transforming the Specifications

In this section we describe how the behavior of the framework is formalized through the use of FTS (Finite Transition System) [Arnold1994]. In the sequence, we show how the specification is described and transformed into reliable code.

5.1 The Behavior of the Framework

Most work on the semantics of parallel, communicating, concurrent or interacting processes is based on the concept of automaton. More generally, a finite state automaton formed of states and labeled transitions between those states, can describe a system whose state evolves over time [Arnold1994]. An agent is a computational entity handling sequences of events. To handle events, agents can emit events, absorb events, and process internal events [Plasil2002]. Method calls on interfaces turn into event, and the architecture's behavior is modeled via the event sequences (traces) on the architecture. The behavior of the architecture can be approximated and represented by FTS. A *transition system* consists of a set of possible states for the system and a set of transitions - or state changes – which the system can effect [Arnold1994].

The previously presented architecture (Figure 3) can be described as a concurrent FTS, as shown in Figure 6. The figure shows each tier represented as a FTS, working concurrently with other tiers. The label λ indicates the target action or event, when the state triggers the transition. The set represented by the states $\{S_1, S_2\}$ encapsulate the provided- and required-interfaces *DomainIn* and *DomainOut* of the Domain tier, respectively. In a similar way, the set $\{S_4, S_5\}$, $\{S_{11}, S_{12}\}$ and $\{S_{81}\}$ compose the provided- and required-interfaces of the MAS, Agent and Infra tiers respectively. The states S_3, S_6, resp. S_{82} represent a set of nested states composed by the classes with the suffixes *Controller*, *Creator* and *Parser* of the Domain, MAS and Infra tiers, as seen on section 3.5.

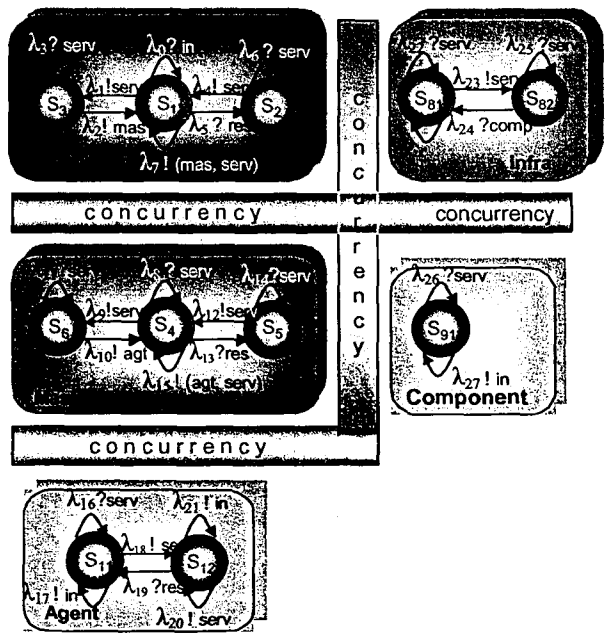

Figure 6 - The architectural model as FTS

Asynchronous behavior between states is represented through self-transition. A self-transition may represent a, asynchronous communication channel between two tiers ((S_1 to S_4, for example) or a recursive decomposition to nested states, as seen on S_3, S_6 e S_{11}. On the expressions that label the transitions, the character λ represent the target action to be executed by the transition. The suffixes {!, ?} represent the action emitted or absorbed. Besides actions, variables are also described. Basically, the variables represent services (*serv*), instances (*mas*, *agt*, and *comp*) and results or data (*res*) modified by the states or processes.

In run-time, the program directs the flow via switch for the current state, evaluates the predicates and changes for the target state, performing the associated action. This can be seen in the code fragment presented on Figure 7 of the next subsection. ECA rules specifies how the architecture receives messages from the environment and from agents, how it verifies the service, direct services, sends messages and create instances of the architectural entities. The synchronism between tiers (considered as concurrent processes) is provided through CCS (Calculus for Communicating Systems) [Milner1985] expressions.

CCS expressions generate a set of *traces* over the architecture and the agents establish the restrictions, the sequence of execution and the synchronism between the concurrent tiers. The basic operators are the classic regular expressions *sequence*, *alternative* and *repetition*. The *enhanced operators* provide a notation to describe concurrency, using the known operators *or-parallel*, *and-parallel* and *restriction*. Several transitions can have the same source and target, i.e., the product mapping is not necessarily injective. The sequence of actions $S(c) = \lambda(t_1)\ \lambda(t_2)$ is called the trace of the path. Intuitively, the label of a transition indicates the action or the event, which triggers the transition.

5.2 Code Generation

When instancing MASs, agents or Infra components, the specifications captured and stored in XML file are transformed into reliable code using parser and generator programs. The parsers can read the specifications from the XML file using the standard XML document object model (DOM). DOM essentially maps every element of an XML document to an object. Such an object has methods to access the element's attributes, and DOM also supplies methods to navigate through documents and to locate the parent element and enumerate the child elements. After being parsed through the DOM, the information is supplied to the generator program, which transforms the parsed information into source code based on templates of MAS-CF entities.

During the implementation phase, code generation occurs at two separate times. First upon the instantiation of the architectural elements by the framework, when the code of the structural model is automatically generated. At this stage, the MAS (if it has not been instantiated), the agents and the internal layers of the agents can be instantiated. Afterwards, only the abstract method of semi-finished component can be implemented or plugged. Thus, the implementation of the internal architecture of the agent becomes independent from the framework. The internal implementation of the agents is free, and therefore any type of agent architecture or implementation model may be used.

In the design of rational agents, the role played by attitudes such as beliefs, desires (or goals) and intentions have been well recognized in the AI and agents literature. Systems and formalisms that give primary importance to intentions are often referred to as BDI (Belief, Desire, Intention) architectures. BDI-like architectures model the

agent's behavior using a set of mental categories evolving in a mental cycle that allows the agent to make decisions and to act on the environment. These architectures raise from the process of deciding, moment by moment, which action to take towards its objectives.

Figure 7 shows a partial view of the generated Java code for the *Mas* (here *Mas* is an instance of the abstract model MAS) class. The interface *MasIn* (line 32), the parameters and the pre-condition (line 34) are supplied from the specification of the interface and the remaining items - states, transitions and actions - can be retrieved from the XML *service* specification. On line 36, the method *run()* of the library class *getState* retrieves the current state of this specific instance. Line 38 performs the transition via *switch* for the *case* that corresponds to the current state. Inside each *case*, the method *instanciaAgent*() of the abstract factory *MasController* is called and returns the instance responsible for forwarding or executing the requested service. On line 44, the target state is defined and stored using the method *run*() of our library class *setState* (line 45). On line 46, the agent returned in the *frame* instance performs the action associated with the transition.

```
24  private
25      final static int MasServiceReceive = 6;
26      final static int MasServiceRequest = 5;
27      final static int DomainServiceReceive = 1;        --> states
28      final static int DomainServiceRequest = 2;
29      final static int AgentServiceReceive = 10;
30      final static int InfraServiceRequest = 20;
31
32  public void MasIn(String service, Vector in, Vector out)  --> interface
33  {
34      if ((service.length()>0)&&(in!=null))             --> pre condition
35      {
36          int state = getState.run();                   --> current state
37
38          switch (state) {                              --> transition
39              case MasServiceReceive:
40                  try
41                  {
42                      IFrameCreator frame = (IFrameCreator)
43                          MasController.instanciaAgente(service, in, out);
44                      state = AgentServiceReceive;      --> target state
45                      setState.run(state);
46                      frame.run(service, in, out);      --> action
47                      break;
48                  }
49                  catch (Exception e)
50                  {   showMessage.run(e);
51                      break;
52                  }
```

Figure 7 - Partial view of the generated code for the Mas class

The code of the Mas class presented above is almost completely frozen (except the name of the interface *In* - *MasIn* - on line 32, the name of the interface *Out* – *MasOut* - and the class name are hot-spots). It is completely generated when elements of the framework are instanced for the first time. The same happens for the classes Domain (through which different domains can be instanced) and Infra. The framework also generate the code for the abstract classes Agent and Component every time new

agents or components are instanced. Specific implementation can be added on the hot-spots provided by the abstract classes of the last level.

We argue that the reuse of organizational abstractions, as well as the interaction facilities provided by the architecture reduces the complexity and facilitates the development of the cognitive capacities of the agents (learning and autonomy), since complex properties such as interaction, adaptation and collaboration can be addressed separately by the architecture. In this fashion, agent implementation can be better focused on the maintenance of its structures of knowledge gathering and on its mechanisms of learning.

6 Discussion and Related Works

The concept of connection as an architectural entity was established on the first ADLs, such as Darwin [Magee1997], UniCon [Shaw-Garlan1996], Wright [Allen1997] and ACME [Garlang1997] among others. The idea is to deal with aspects and system qualities in connectors, not in components. According to Szyperski [Szyperski2002], one of the problems with these approaches is that by introducing a pure connection-oriented approach, all components are restricted to only interact with other components if appropriately connected. On the other hand, a connector, when detailed, can easily heave substantial complexity and display a need to be partitioned into components itself. Thus, "connectors" turn into regular components and no special actions can be performed on the connections as such.

The concept of explicit connector has been loosing ground as time passes. Some ADLs, such as Rapide, 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. Faulkner [Faulkner2001] proposed an ADL for multi agent systems using a similar concept. In his approach, Faulkner uses components, interfaces and services as architectural entities, without connectors. Connections are implemented as bindings between provided interfaces and services. Szyperski [Szyperski2002] states "contextual component frameworks can be used to reintroduce the intercepting behavior of connectors, but this time at the level of context boundaries." Contexts provide the generic-aspects, while components and/or agents provide the non-generic aspects of contexts by parametrizing generic contexts.

Our approach has a very weak notion of connector. The interaction rules are managed and performed by the architecture, resulting in calls to the other agents and services inside or outside of the organization. Its semantics consists of the rules defining the subtype (and supertype) relationship between tiers, and the services ontology providing the necessary mechanisms to interoperability support. Wooldridge [Wooldridge2000] states that agents are not built considering the existence of other specific agents; the idea is that interdependencies are likely to be reduced to make the system more flexible and reusable.

The preference for implicit connections, as opposed to explicit ones, is one of the key points in our approach, using a very weak notion of connector. Interaction rules are regulated and executed by the architecture, resulting in calls to other agents and components inside and outside the organization. The semantics consists of rules defining the relationship between superior and inferior layers and the ontology service providing support mechanisms necessary to interoperability. We share a concept introduced in [Wooldridge2000], whereas agents should not be built assuming the existence of other specific agents; the idea is that interdependencies may be reduced to make the systems more flexible and reusable.

Current frameworks for multi-agent systems such as JADE [Bellifemine2001], RETSINA [Sycara1999, Sycara2003], MESSAGE [Evans2002] and ZEUS [Azarmi2000] work with a structure much more focused on the individual properties of agents than on MAS architecture. These approaches provide an implementation that reinforces only partially the rules of interaction in the architecture. Unlike most frameworks for multi-agent systems, our framework focuses on the reuse of generic abstractional organizations instead on the individual agent properties such as roles, protocols and internal architecture.

7 Contribution and Practical Results

Our key contribution is to describe a MAS in an abstract and application-independent way, allowing large-scale reuse of the abstractional organizations. We were able to show, throughout the work, the support to architectural principles and the use of contextual compositions, allowing the reinforcement or solution at an architectural level, of some of the fundamental agency properties cited on Section 2 such as *interaction*, *adaptation* and *collaboration*. This makes the implementation of the agent much simpler since such aspects are addressed separately from the object's functional implementation. The following properties were directly or indirectly addressed at an architectural level:

- *interaction*: the rules of interaction established by the communication model forcing the instance of an agent to communicate via a control mechanism of the architecture makes possible the distribution and sharing of services in a transparent and independent way.
- *adaptation*: the abstract factories of the Domain, MAS and Infra tiers allow new agents or new version of agents replacing obsolete ones to be easily "plugged" in our framework, ensuring high flexibility and adaptability since the agents can easily adapt its state and behavior in run-time to new environment conditions.
- *collaboration*: the formalization of services through ontologies and catalogs communicate the semantics of the services provided by the agents and generic components, facilitating the assembly of composition and collaboration between agents via required- and provide-services. Forcing all agents to use a common vocabulary defined in one or more shared ontologies

is an oversimplified solution especially when these agents are designed and deployed independently from each other.

Reusing an abstract architecture allows the reuse of not only architectural software design and implementation, but also of some agent properties that can be controlled via architecture mechanisms. Those benefits allow large-scale reuse reducing the time of system development and for system readiness.

We have instantiated a medical application for behavioral therapy using our framework. We were able to verify the facilities provided by the framework and at the same time evaluate certain non-functional requirements such as applicability, usability and performance among others. The system, called *MAS-CF Therapp* [Caminada2004] provides services for a larger application that uses Virtual Reality on the therapy of autistic children and children with a psychosis diagnosis. The system works in a distributed web environment, through the HTTP and TCP/IP protocols using *Java/JSP/Servlet* technology in conjunction with a *Java/Tomcat* server.

For the first time our MAS-CF framework could be evaluated in a real world application. From the viewpoint of practical applicability and use of the described techniques, the following could be evaluated:

- the contextual paradigm tiers of MAS-CF;
- the interaction model used by the framework;
- the viability of using MAS as well as the interaction with Virtual Reality techniques in such a way as to aid and support behavior therapy.

During the development process we could verify the advantages provided by the MAS-CF framework. The implementation of the agents was widely facilitated since the development was concentrated solely on the services provided and the relationships between layers necessary to providing these services. More concrete results will be obtained from future applications to be instantiated.

Acknowledgement

I would like to thank Professor Carlos Lucena for his contribution in this work.

The Brazilian Ministry of Science and Technology provides financial support to this research work through CNPq grants nº 140604/2001-4.

References

[**Adam2004**] Adam E. and Mandiau R. "Design of a MAS into a Human Organization: Application to an Information Multi-Agent System." In Proc. 5th Agent-Oriented Information Systems, pages 1-15, Chicago, IL, USA, October 13, 2003. Springer Verlag, LNAI 3030, 2004.

[**Allen1997**] R. Allen and G. Garlan. "Formalizing Architectural Connection." In Proc. 16th International Conference on Software Engineering, pages 71-80, Sorrento, Italy, May 1997.

[**Amandi1997**] Amandi A.A. "Programação de Agentes Orientada a Objetos". CPGCC UFRGS – Tese de Doutorado, Porto Alegre, 1997.

[**Arnold94**] Arnold A. "Finite Transition Systems". PrenticeHall, Masson, Paris, 1994.

[**Azarmi2000**] Azarmi N., Thompson S. "ZEUS: A Toolkit for Building Multi-Agent Systems". Proceedings of Fifth Annual Embracing Complexity Conference, Paris April 2000.

[**Baral997**] Baral, C., Lobo J., Trajcevski G. "Formalizing Workflows as Collections of Condition-Action Rules". Dept of Computer Science, University of Texas at El Paso. El Paso, Texas, USA,1997.

[**Bellifemine2001**] Fabio Bellifemine, Agostino Poggi, Giovanni Rimassa: JADE: a FIPA2000 compliant agent development environment. Agents 2001: 216-217.

[**Berners-Lee2001**] Berners-Lee, T.; Lassila, O. Hendler, J. – The Semantic Web – Scientific American - http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html

[**Bond1988**] Bond A.H. et al. "Readings in Distributed Artificial Intelligence." San Mateo, Morgan and Kaufmann, 1988.

[**Bosch1999**] Bosch, J., Molin P., Mattsson M.; Bengtsson P.; Fayad M. "Framework problem and experiences" in M. Fayad, Building Application Frameworks, John Willey and Sons, p. 55–82, 1999.

[**Breitman2004**] Breitman K, Haendchen Filho A., Haeusler E. H., Staa, A. V. "Using Ontologies to Formalize Service Specification in Multi-Agent Systems". Proceedings Of Third NASA/ IEEE Workshop on Formal Approaches to Agent Based Systems, Los Alamitos, California USA. To appear as LNCS, Springer-Verlag.

[**Brussel1999**] Hendrik Van Brussel , Jo Wyns , Paul Valckenaers , Luc Bongaerts , Patrick Peeters, Reference architecture for holonic manufacturing systems: PROSA, Computers in Industry, v.37 n.3, p.255-274, Nov. 1998.

[**Caminada2004**] Caminada, Nuno. "Uma Aplicação Terapêutica de Realidade Virtual Utilizando Tecnologia Baseada em Agentes de Software". Projeto Final de Conclusão do curso de Graduação em Ciência da Computação. UniverCidade – Unidade Ipanema. Junho 2004, Rio de Janeiro, Brasil.

[**Connolly2000**] Connolly D. "Extensible Markup Language (XML)." February 2000. Available on-line: http://www.w3.org.XML/.

[**Correa1994**] Correa Filho M. "A Arquitetura de Diálogos entre Agentes Cognitivos Distribuídos". COPPE da UFRJ. Tese de Doutorado, 1994.

[**Dashofy2001**] Dashofy, E.M.,Hoek, A.v.d., and Taylor, R.N. "A Highly-Extensible, XML-Based Architecture Description Language." In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). Amsterdam, The Netherlands, August 28-31, 2001.

[**Evans2000**] Evans R. (Editor). "MESSAGE: Methodology for Engineering Systems of Software Agents". Deliverable 1, July 2000.

[**Faulkner2001**] Faulkner S. "Towards an Agent Architectural Description Language for Information Systems". Technical Report, University of Louvain, Belgium, 2001.

[**Fayad1999**] Fayad M.E. et al. "Building Application Frameworks". John Wiley & Sons, Inc. New York, 1999.

[**Fensel2003**] Fensel, D.; Wahlster, W.; Berners-Lee, T.; editors – "Spinning the Semantic Web" – MIT Press, Cambridge Massachusetts, 2003.

[**Finin1997**] Finin T. "KQML as an agent communication language." Proceedings of the Third International Conference on Information and Knowledge Management". CIKM-94, ACM Press, november 1994.

[**FIPA1997**] Reference FIPA-OS V2.1.0. Nortel Networks Corporation, Ontario, Canada, 2000. FIPA-OS site http://www.emorphia.com/home.htm.

[**Gamma1995**] Gamma E. et al. "Design patterns – elements of reusable object-oriented software." Addison-Wesley Longman, Inc., 1995.

[**Garcia2003**] Garcia A., Lucena C.J.P. et al. (Eds.) "Software Engineering for Large-Scale Multi-Agent Systems". Lecture Notes in Computer Science – LNCS 2603, Springer Verlag, Germany, 2003.

[**Garcia2001**] Garcia A., Torres V. In: "Sistemas Multi-Agentes". Editores: Carlos Lucena e Ruy Milidiú. Editora Papel Virtual, Rio de Janeiro, 2001.

[**Garlan1997**] D. Garlan, R. T. Monroe, and D. Wile. "ACME: An architecture description interchange language." In Proc. CASCON'97, pages 169-183, Toronto, Canada, Nov. 1997.

[**Gruber1998**] Gruber, T. – "A translation approach to portable ontology specifications." Knowledge Acquisition, 5(2):21--66, 1998.

[**Griss2003**] Griss M.L., Kessler R.R. "Achieving the Promise of Reuse with Agent Component." Software Engineering for Large-Scale Multi-Agent Systems. Springer, LNCS 2603, Germany, 2003.

[**Haendchen2004**] Haendchen Filho, A.; Staa, A.v.; Lucena, C.J.P. "A Component-Based Model for Building Reliable Multi-Agent Systems". Proceedings of 28th SEW - NASA/IEEE Software Engineering Workshop, Greenbelt, MD. IEEE Computer Society Press, Los Alamitos, CA, 2004, pg 41-50.

[**Holvoet2003**] T. Holvoet and E. Steegmans. "Application-Specific Reuse of Agen Roles." Software Engineering for Large-Scale Multi-Agent Systems. Springer, LNCS 2603, Germany, 2003.

[**Kendall1999**] Kendall E. et al. "A Framework for Agent Systems". In: Implementing Applications Frameworks – Object-Oriented Frameworks at Work. M.Fayadd et al. John Wiley & Soons, 1999.

[**Kotak2003**] Kotak D. et al. "Agent-based holonic design and operations environment for distributed manufacturing". In: Computer in Industry, Volume 52, Issue 2, pg 95-108 – Elsevier Science Publishers B. V. Amsterdam, The Netherlands, 2003.

[**Kwangyeol2003**] Kwangyeol R. et al. "Agent-Based Fractal Architecture and Modelling for Developing Distributed Manufacturing Systems." International Journal of Production Research, vol. 42, N0. 17 (2003) 4233-4255.

[**Larman1997**] Craig Larman . "Applying UML and Patterns." Prentice Hall PTR, New Jersey, 1997.

[**Magee1999**] Magee J., Kramer J. and Giannakopoulou D. "Behaviour Analysis of Software Architectures," presented at the 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, 22-24 February 1999.

[**Milner1985**] Milner R. "Lectures on a Calculus for Communicating Systems". Lectures Notes in Computer Science, Vol. 197 - Springer Verlag, 1985.

[**Montesco2001**] Montesco C.A.E. et al. "UCL – Universal Communication Language". Universidade de São Paulo, Instituto de Ciências Matemáticas e da Computação. Technical Report, São Paulo, Brasil, 2002.

[**Noriega1997**] P. Noriega. "Agent-mediated Auctions: THE Fiskmarked Metaphor." PHD Thesis, Universitat Autonoma de Barcelona, Barcelona, 1997.

[**OMG2000**] Object Management Group. Agent Platform Special Interest Group. "Agent Technology – Green Paper", version 1.0, September 2000.

[**Plasil2002**] Plasil F. et al. "Behavior Protocols for Software Components". IEEE Transactions on Software Engineering, Vol. 28, N. 11, November 2002.

[**Pree99**] Pree, W. "Hot-spot-driven development" in M. Fayad, R. Johnson, D. Schmidt. Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Willey & Sons, 1999.

[**Sodhi2000**] Sodhi, J. et al. Software Reuse - Domain Analysis and Design Process. New York: McGraw Hill, 1998. 344 p.

[**Srinivasan2001**] Srinivasan P. "An Introduction to Microsoft .NET Remoting Framework". Microsoft Corporation, July 2001.

[**Sycara1999**] Sycara, K. "In-Context Information Management through Adaptative Collaboration of Intelligent Agents". Intelligent Information Agents. Edited by Matthias Klusch. Springer-Verlag, Berlin, 1999.

[**Sycara2003**] Sycara K. et al. "The RETSINA MAS, a Case Study." Software Engineering for Large-Scale Multi-Agent Systems". Lecture Notes in Computer Science – LNCS 2603, Springer Verlag, Germany, 2003, p. 232-250.

[**Szyperski2002**] Szyperski C. "Component Software – Beyond Object-Oriented Programming." Addison-Wesley and ACM Press, 2000.

[**Todd2003**] Todd N., Szolkowski M. "Java Server Pages". Elsevier Ed., 2003.

[**Tracz1994**] W. Tracz. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). ACM Software Engineering Notes, 19(2):52-56, Apr. 1994.

[**Vazquez2003**] Vazquez J. "The HARMONIA framework: the role of norms and electronic institutions in multi-agent systems applied to complex domains." Technical University of Catalonia, Barcelona, Spain. ISSN 0921-7126, IOS Press, 2003.

[**Vitaglione2002**] Vitaglione G., Quarta F., Cortese E. "Scalability and Performance of JADE Message Transport System". Proceedings of AAMAS Workshop on AgentCities, Bologna, 16th July, 2002.

[**Zambonelli2002**] Zambonelli, F., Jennings, N.R., Wooldridge M. Organisational Abstractions for the Analysis and Design of Multi-agent Systems. In P. Ciancarini, M.J. Wooldridge, AgentOriented Software Engineering, vol. 1957 LNCS, 235-251. Springer-Verlag: Berlin, Germany 2001.

[**Wooldridge2000**] Wooldridge M., Jennings N. and Kinny D. "The Gaia Methodology for Agent-Oriented Analysis and Design". Proceedings of 3rd International Conference on Autonomous Agents, Seatle, WA, 1999.