A Framework-Based Approach for Building Reliable Multi-Agent Systems

Aluízio Haendchen Filho **Arndt von Staa** **Carlos J.P. Lucena**

aluizio@inf.puc-rio.br, arndt@inf.puc-rio.br, lucena@inf.puc-rio.br

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente 225, Gávea
22453-900 Rio de Janeiro, RJ, Brasil

PUC-Rio.Inf.MCC22/03 Agosto/2003

Abstract In this article we present a framework-based approach for rapid and large-scale development of reliable and reusable multi-agent systems (MAS). The agents are instanced through a specification framework and the reuse is achieved by the structural model propagation for multiples agents. Flexibility and agent adaptation capacity is ensured through the use of design patterns foundations, such as encapsulation, high-cohesion, low-coupling - and extensible structures. Quality control is based on three fundamentals: (i) formal specification; (ii) inspections; and (iii) testing. Formal systems specification leads to codes with less errors, and rigorous processes of verification and validation replace the mathematical proofs. The system reliability is assured by the testing tools and facilities provided to perform semi-automated inspections and test cases generation, applied together with an incremental testing strategy. The purpose is reducing time, effort and costs associated with the design and development of MAS with high requirements of quality and reliability.

Keywords: multi-agent systems, agent reliability, agent framework, MAS quality control

Resumo: Neste artigo apresentamos uma abordagem baseada em framework para desenvolvimento rápido em larga escala de sistemas multi-agentes confiáveis e reusáveis. Os agentes são instanciados através de um framework de especificação e o reuso é alcançado pela propagação do modelo estrutural para múltiplos agentes. Flexibilidade e capacidade de adaptação são garantidas pela utilização de fundamentos de design patterns, tais como encapsulamento, alta-coesão e baixo-acoplamento – e estraturas extensíveis. O controle de qualidade é fundamentado em três requisitos: (i) especificação formal; (ii) inspeções e (iii) teste. A especificação formal de sistemas conduz à produção de código com menos erros, e rigorosos processos de verificação e validação substituem as provas matemáticas. A confiabilidade do sistema é assegura por ferramentas de teste e pelass facilidades providas para executar inspeções automatizadas e geração de casos de teste, aplicadas junto com uma estratégia de teste incremental. O objetivo é reduzir tempo, esforço e custos associados ao projeto e desenvolvimento de sistemas multi-agentes com altos requisitos de qualidade e confiabilidade.

Palavras chave: sistemas multi-agentes, confiabilidade de sistemas-multi-agentes, framework para sistemas multi-agentes, controle de qualidade de sistemas multi-agentes

Summary

1. INTRODUCTION **1**

2. PROPOSED APPROACH **1**

3. SPECIFICATION MODEL **2**

3.1. STRUCTURAL MODEL 3
3.2. INTERFACE SPECIFICATION 5
3.3. BEHAVIOR MODEL 6

4. REFINED SPECIFICATION AND PROTOCOL COMPLIANCE **8**

5. MAS EVALUATING **9**

5.1. TESTING STRATEGY 9
5.2. VERIFICATION AND VALIDATION 10

6. RELATED WORKS **12**

7. CONCLUSION AND ONGOING WORKS **13**

8. REFERENCES **14**

1. Introduction

Questions related to multi-agent systems' (MAS) building, verification and validation are among the most important fields driving research in agent-oriented software engineering nowadays. MAS operate on highly dynamic form, where a multiplicity of external systems, services, and users interact and change the environment [2] [18]. The communication model is usually asynchronous and, in addition, there is no pre-defined flow from an agent to another. An agent can autonomously begin an internal or external behavior at any time [34], not just when a message is sent. Each of these properties introduces additional complexity and increases the possibility of imperfections and manifestations of exceptions. Some questions can be raised in this context: (i) considering the complexity, highly dynamic, and evolving characteristics of the agents, which software engineering techniques are most appropriate for development and building of MAS?; (ii) during the development and evolution process, how can we evaluate MAS reliability and be sure about its conformity with the requirements and adopted patterns [3] [9] [16] ?

Components and agents have been described in literature as abstractions possessing aspects of close similarity [4] [14] [22]. Agents have been considered as the next generation components [14], which in turn support the development of flexible and evolving applications, such as those behind ecommerce and web-services. Agent-oriented software engineering extends the conventional components' development approach, leading to the construction of more flexible and component-based MAS [14], emphasizing reuse, low-coupling, high-cohesion and support for dynamic compositions. Rapid and problem-specific system construction can be attained through the use of model-driven development and reuse techniques in order to make the application more flexible, adaptable, robust and self-managing. These properties can be composed by the combination of several technologies, such as component-based software engineering, frameworks [7], design patterns [12] [20] and rule-based systems [13] [24] . These concerns are explored by Griss [14], who proposes the integration of these areas in order to build flexible and reusable MAS.

In this paper we describe a framework-based approach that integrates different software engineering technologies for design and development of MAS with high requirements of quality and reliability. We define framework from an object-oriented design perspective [7]. More specifically, a framework is a set of cooperating classes that allows a reusable design for a specific domain. A developer customizes the framework to a particular application by extending and composing instances of framework classes. The implementation of our framework is currently being developed. Departing from the proposed model, we have built several modules that compose the framework. Those modules perform several of the key activities foreseen in our approach. Reuse is achieved through the departure from a structural model, implemented using design patterns, such as *command*, *mediator*, *singleton*, *builder* and *facade* (GoF) [12]. The utilized design patterns emphasize the proprieties of high-cohesion and low-coupling, providing support for dynamic compositions, flexibility and extensibility to MAS.

2. Proposed approach

Our main purpose is to develop a framework and tools that can be used to aid in a rapid construction of flexible and reusable MAS. The framework is constructed departing from two models: a specification and a test model. The specification model is composed by a set of

representations, which support evolution and reuse based on flexible structures, such as micro-frameworks and design patterns. The test model provides a group of classes to support the verification and validation processes. The verification of specification compliance is done during the design phase by using static automated inspections. A set of evaluation tools facilitates the creation of scenarios for test cases at run-time. The conditions for testing the adherence of agents to the specifications are performed by state-based testing. The quality control is based upon four activities: (i) formalization of the design models; (ii) automated inspections; (iii) instrumentation and (iv) testing.

Figure 1 shows the main elements of the generic architecture. Each model describes the subsystems and their relationships. The different tonalities distinguish the functionality of each module. The dark-gray boxes (*Goals*, *Specification Model*, *Test Model* and *Control Model*) represent models. The light-gray boxes (*Agent*, *Modeling Framework*, *Testing Framework* and *Test Artifacts*) represent program code. Off-line inspections are performed using documented criteria [16] and are represented by dotted lines.

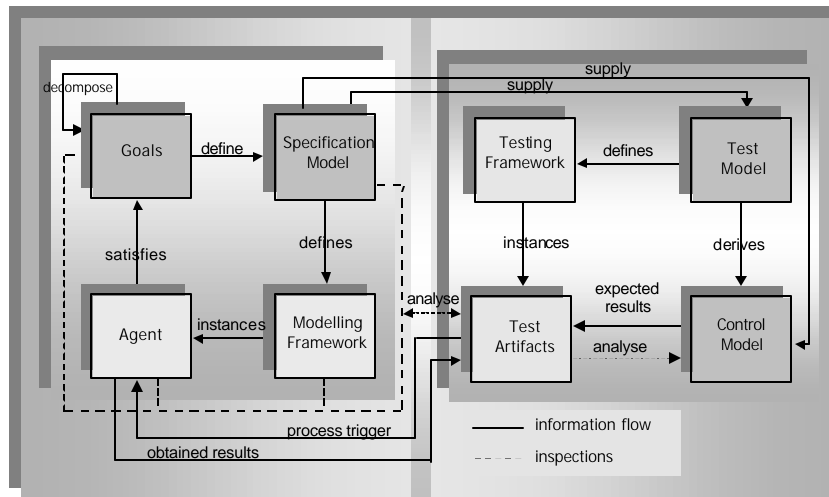

Figure 1 - Generic architecture

The *Specification Model* identifies the architecture and system behavior and defines a *Modeling Framework* that provides a set of classes which can be extended to compose instances of agents. The agents are instantiated through an interactive and semi-automatic process supported by graphical interfaces. The *Test Model* identifies a set of models that defines the *Testing Framework*. Testing framework provides mechanisms and tools for the test artifacts development and instantiation. *Test Artifacts* include all the testware artifacts generated during the test, such as test scripts, test drivers, test suites, automated inspections and so on. The *Control Model* is the base for state-based testing; it defines an oracle that can be generated after refined specifications have been made.

3. Specification model

The specification model is composed by a group of abstractions described by four basic representations: (i) *structural model*, which deals with establishing a basic structural framework for MAS; it encompasses identification of components, subcomponents or modules and classes that compound the framework and its relationships; (ii) *interface model*,

which defines a model for interface specifications, similar to IDL CORBA; (iii) *behavior model*, that formally describes the behavior of the agents via behavior protocols [22] [23] [28] and (iiii) *refined specification*, which applies state-based specifications and rule-based system for compliance verification.

3.1. Structural model

The initial design process consists in identifying MAS architecture, subsystems or agents supplying a range of related group of services. Large scale reutilization is provided by the architecture, since the structural model can be propagated to systems and components with similar requirements. In order to create an instance of an application in our framework, we utilize a simple MAS, which tracks undergraduate student advisement process. Figure 2 shows the partial Electronic Advisement MAS generic architecture, composed by five layers: *Secretary*, *Advisor*, *Chair*, *Instructor* and *Infrastructure*. A student starts the process sending a web formulary requesting advisement. After receiving the request, the secretary analyses it and forwards to the *Advisor*. The agents *Instructor* and *Chair* exchange messages with human agents by well-defined and well-structured e-mail messages.

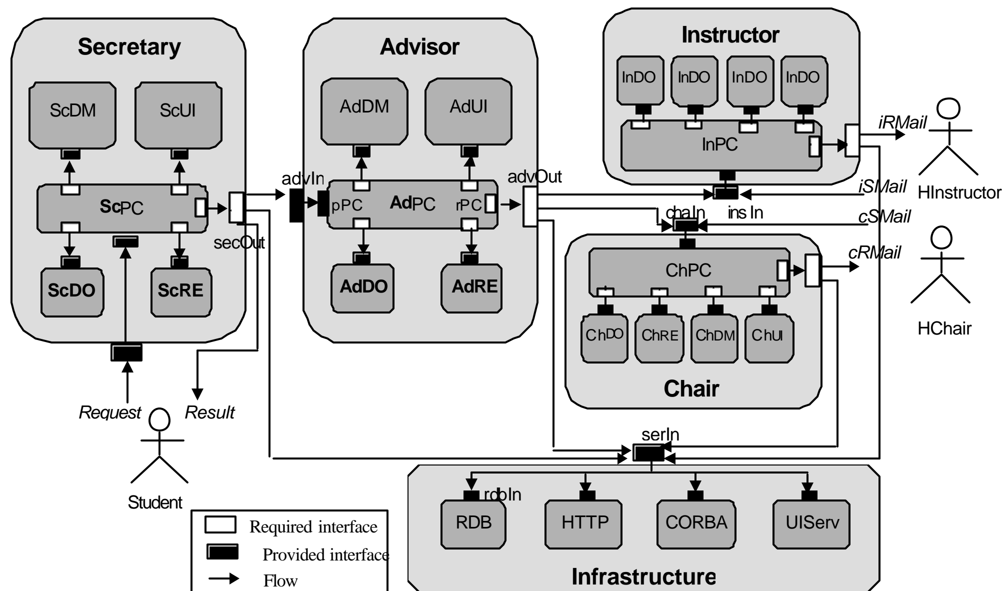

Figure 2 - MAS structural model

The agents' internal structure can hold five layers: (i) *Perception* & *Communication* (PC), used to retrieve environment information and interact with other agents; (ii) *Decision & Management* (DM), which controls the lifecycle of the agents and actions that need performed to be performed to make decisions; (iii) *Domain* (DO) groups methods that perform domain application dependent tasks; (iv) *Resources* (RE), that includes resources needed by the agent, such database access drivers and (v) *User Interface* (UI), where graphical interfaces objects utilized by the agents to interact with the environment can be specified. This structure allows distinguishing cognitive and reactive agents. The reactive agents are normally responsible for information retrieval or execution of specific tasks. It generally possesses only three or four of the five layers. Each layer can have nested subcomponents. The agents'

subcomponents contain the initials of the internal layers as suffix: DM for *Decision & Management*, RE for *Resources*, DO for *Domain* and PC for *Perception & Communication*.

The *Infrastructure* layer defines a framework for the classes responsible for providing generic services such as communication with remote servers via RMI/CORBA, access the local database, Web pages, and assembly of graphical interfaces, normally required for the domain applications classes. These services may or may not be used by the agents. At times, in order to make the agent less dependent, it may be of advantage configures them to incorporate specific services in its internal layer *Resource* (…*RE*), preventing the coupling with the services layer and preserving the agent autonomy to work in other platforms. The communication services provided are not limited to basic support for communication; they can also provide generic communication resources such as dial-up connections and POP/SMTP services.

The structural model is broken down using object-oriented techniques, and establishes a set of standards that define basic responsibilities to the classes that compose the framework. In our model we use five design patterns defined by Gamma et al. [12]: *Command*, *Facade*, *Mediator*, *Singleton* and *Builder*. The attributions of responsibilities to the framework classes had been defined by Larman [20]: *Controller*, *Creator*, *Low-Coupling* and *High-Cohesion*. Each layer possesses only one provided-interface, where the environment requests and agent's calls come through; and one required-interface, for which the agent can be disponibilize services and to communicate with the environment or anothers agents. The interface receives the calls and delegates to the *mediators* classes *PC*, who will decode the call and coordinate the processing of internal flow. It is important to point out that the implementation of the GoF pattern *Mediator* [12], used in the *PC* classes (*AdPC*, *SePC*, *InPC* and *ChPC*) also works as a GRASP pattern *Controller* and *Creator* [20], which coordinate and instantiate the internal classes of the agent for the requested service execution. This model of construction also assures that the architecture is *plug-and-play*, can evolve and new functionalities can be enclosed, since all this can be achieved by modifying only the mediators.

The *Infrastructure* layer is modeled as a superclass that receives requests from services via its mediator interface *serviceIn*. The mediator evaluates the parameters and instantiate the classe whose service is required. New services can be pluged in this layer through modifications of the mediator *serviceIn*. The use of the *GoF* pattern *Builder* in the *UiServ* class centers the process of the creation of graphical interfaces through a specialized task that returns the GUI interface in the output vector. The *UiServ* class works as a builder, receiving in the input vector graphical components already formatted by the respective classes *RE* of the agent' internal structure, and returns the required interface in the output vector. Some classes in this layer, such as *RDB* and *HTTP* implement the GoF pattern *Singleton*, which limits the number of connections. That is achieved departing from the creation of a single instance of these classes that controls a stack of connections to the servers, and consequently optimizes the memory alocation.

To formalize the MAS architecture, we use a CDL (Component Definition Language) similar to ADLs (Architecture Definition Languages) [28] and based on SOFA notation [22] [29]. Two steps describe the structure of implementation: (i) instantiation of direct subcomponents and, (ii) specification of the connections between the subcomponents and components via interface ties. In the first step, each of the subcomponents must be specified for its abstraction level frame. The specification is made through a construct frame, which encapsulates instances of the *provided* and *required-interfaces*. The architecture of the frame *Advisor* (Table 1) illustrates how the subcomponents are instantiated and how its ties are specified in the construct frame. Reflecting a top-down design, the specification of an application is factored in alternate levels: *frame* – *architecture* - *frame* -, forming a tree with nodes of the alternating

type frame and architecture [22]. The frame concept corresponds to a primitive agent, and the architecture concept corresponds to its first level of decomposition.

Frame Advisor **{**	
provides:	**delegate** Advisor: advIn **to** AdPC: pPC;
Advisor advIn;	**subsume** AdPC: rPC **to** Advisor: advOut;
requires:	
Instructor instrIn;	};
Chair chaIn;	**Frame** AdPC {
Service serIn;	**provides**:
};	AdPC pPC;
Architecture Advisor {	**requires**:
Inst AdPC Apc;	AdRE r1PC;
Inst AdDM Adm;	AdDO pDO;
Inst AdRE Are;	...
Inst AdDO Ado;	};
Inst AdDO Aui;	

Table 1 - Partial example of the architecture formal specification for the agent Advisor

The second step establishes the relationship model. The agents publish their services by means of *provided-interfaces* and request services through its *required-interfaces*. We distinguish three different types of ties between interfaces [22]: (i) *bind* expresses the tie of a *required-interface* to a *provided-interface* between two subcomponents (e.g. *secOut* and *advIn*); (ii) *subsume* defines the tie from a subcomponents' *required-interface* to a components' *required-interface* (e.g. *rPC* and *advOut*) and (iii) *delegate* identifies the tie between a *provided-interface* of the component and a subcomponent *provided-interface* (e.g. *advIn* and *pPC*). This distinction is important because the *delegate* type [33] points out to inherence implementation and the *subsume* type contrives the notion of treating the split object as a whole, maintaining encapsulation for the whole. The formalization of the structural model defines a standard textual representation that can be used to build tools for automated testing and for the verification of the conformity between the original project and the implemented code.

3.2. Interface specification

The key mechanism making MAS highly flexible - although initially more complex -, is the agents' interaction model [18] [34]. Instead of interacting with multiple interfaces as it is usually done in component models, the agents preferentially receive requirements trough a single interface, using highly structured messages. These messages can be extended as the system evolves, making the agent components flexible and highly reusable. An interface type is a set of method signatures composed by a syntactic description of each method, parameters name, return values and possible exceptions, as shown in Table 2.

The definitions of interface type are expressed via interface construct, whose syntax is quite similar to the IDL CORBA notation. The example in Table 2 shows the *rdbIn* interface - of the *Infrastructure* class - that receives request to manipulate information in the database. The Mediator *serIn* passes a set of parameters through a vector whenever it calls a method of the *RDB* class. The first element contains information on which service is requested. In accordance with the first parameter, the interface decides if the service is to insert new data in the database, to retrieve some information, to open or to close a database. The mapping of the attributes for the database, as well as the JBDC driver is also configured in this stage. The interface model contains a syntactic description of each method, its parameter names, types, return values and possible exceptions.

Interface type	Method	Input parameters	Output parameters	Return parameters	Exceptions
rdbIn	dbGetInstance	in String database		dbServ db	SQLException
	void dbSetSQL	in Vector entrada			SQLException
	void dbGetSQL	in Vector entrada	out Vector saida		SQLException
	void open	in String database in String path			SQLException
	void close	in String database in String path			SQLException
...	...				

Table 2 - Interface type specification

In order to complement the syntax of the interface, it is necessary to identify the axioms [27] [30] that define the conditions that are always true for different combination of operations, as well as the restrictions applied for each operation. In a similar way, the pre and post-conditions that define the valid states for the transitions must be specified. Pre-conditions are conditions that the operation caller agrees to satisfy. In other words, they are conditions that need to be true before executing the method. Post-conditions are conditions that need to be true after the method has been executed. These specifications are stored in an interface specification schema and are also used to perform automated inspections.

3.3. Behavior model

Individually, an agent can be seen as a computational entity handling sequences of events. When manipulating events, the agents can produce or absorb external events, as well as process internal events. An external event can be a stimulus received from the environment or a message received from another agent. An internal event is an operation or a message exchanged among the agent's internal modules. To work as agents, the components must be synchronized and controlled, and their services provided at the right place and time. The agent behavior can be described through behavior protocols [22], which establish the restrictions, the sequence of execution and the synchronism for the action sets. Protocols can be seen as sequences of events, and sequence of events can be expressed or represented by *tokens*. Tokens represent ideas and objects of the real world, using characters and symbols for identification.

Two approaches [30] have been used for formal specification and to write detailed specification of non trivial software systems: (i) an *algebraic* approach [47], where the system are described in terms of operations and its relationships, and (ii) a *model-based* approach, where a model of the system is built using mathematical constructions, and the operations of the system are defined based on their modification of the state of the system [30]. We use a *model-based* approach, that supports different languages to model the behavior of formal specification, among which Z [41], B [42], VDM [43], CCS [21], CSP [44], Petri Nets [45].

In our approach we use the SOFA notation [22] [29], because it proved to be better suitable to describe the language of the agents and also because it was applied successfully to specifying agent components [29] [23] [22]. SOFA uses some particular prefixes and suffixes to represent not only events but also the classic basic operators defined by CCS language [21]. Our grammar comprises four distinct sets: *basic operators*, *enhanced operators*, *prefixes* and *suffixes*. The operators' syntax and its semantic meaning are briefly described in Table 3. The *basic operators* are the classic regular expressions (*sequence*, *alternative* and *repetition*) proposed in CCS language. The *enhanced operators* provide a notation to describe concurrency, using the known operators *or-parallel*, *and-parallel* and *restriction*. The event *prefixes* {! , ?} indicate if an event is absorbed or produced, respectively, and reflects the end of the connection from

the point of view of the event. The *suffixes* represented by {?, ?, ~} are used to indicate return solicitation, reply of return and nondeterministic action.

<table>
<tr><td colspan="3">Basic operators CCS (A and B are protocols)</td></tr>
<tr><td>A ; B</td><td>sequencing</td><td>The set of traces formed by concatenation of a trace generated by A and a trace generated by B</td></tr>
<tr><td>A + B</td><td>alternative</td><td>The set of traces which are generated either by A or by B</td></tr>
<tr><td>A?</td><td>repetition</td><td>Equivalent to NULL + A + (A ; A) + (A ; A ; A) + ... where A is repeated any finite number of times</td></tr>
<tr><td colspan="3">Enhanced operators CCS/SOFA</td></tr>
<tr><td>A | B</td><td>and-parallel</td><td>Arbitrar y sequence of event tokens of traces generated by A and B</td></tr>
<tr><td>A || B</td><td>or-parallel</td><td>Stands for A + B + (A | B)</td></tr>
<tr><td>A / G</td><td>restriction</td><td>The event tokens not in a set G are omitted from the traces of A.</td></tr>
<tr><td colspan="3">Prefixes SOFA adapts</td></tr>
<tr><td>? m</td><td colspan="2">Incoming external call (requirement)</td></tr>
<tr><td>! m</td><td colspan="2">Outgoing external call (service)</td></tr>
<tr><td colspan="3">Suffixes SOFA/CCS</td></tr>
<tr><td>IA ?</td><td colspan="2">Remote call with return</td></tr>
<tr><td>IB ?</td><td colspan="2">Reply with return for an performed remote call</td></tr>
<tr><td>IA ~</td><td colspan="2">Non-deterministic choice. The required-interface can not reply the calling</td></tr>
</table>

Table 3 - Operators, prefixes and suffixes based on SOFA and CCS notation

The precedence of the operators is as follows [22]:

1. (highest) repetition (?), restriction (/),
2. sequencing (;),
3. and-parallel (|), or-parallel (||)
4. alternative (+)

A behavior protocol *P* can be formally described by a regular-like expression that generates a set of *traces* over a system [22]. The set of all possible sequences of events of the agent *A* in a system *S* in any run is referred to as the behavior of *A* in *S*. The simplest behavior protocol is an event *token* or the symbol *null* (full trace) [22]. To demonstrate behavior protocols as a tool for behavior language generation, we consider the *advIn* protocol of the agent *Advisor*. It starts an action from a message received from the agent *Secretary*. After receiving the message, the action is initiated and a sequence of events happens between the *Advisor*, *Instructor* and *Chair* agents. The following expression t_g contains the representation of all the possible traces of the protocol *advIn*, using the previously described notation:

t_g - ((?advIn? ; !advOut ; ?serIn) + !advOut ?~ (? insIn? || ?chaIn?))

Three possible traces can be derived from the t_g expression:

t_1 - ?advIn? ; !advOut ; ?serIn ; !advOut ?~ ; ? insIn?

t_2 - ?advIn? ; !advOut ; ?serIn ; !advOut ?~ ; ?chaIn?

t_3 - ?advIn? ; !advOut ; ?serIn

where: $\{t_1, t_2, t_3\} \subseteq t_g$

Each of the *traces* t_1, t_2, t_3 defines a sequence of calls. The *Advisor* agent receives a stimulus coming from another agent (*Secretary*), notifying the arrival of a new advisement request. After verifying the data, and depending on the state of the variables *class* and *prereq*, one among the three traces t_1, t_2, t_3 must be performed. On *trace* t1 the student meets all prerequisites to be enrolled on the requested disciplines and the class is not full. On *traces* t_2 e t_3 the student does not meet some or all the prerequisites. In this case, an authorization is solicited through the agents *Instructor* and/or *Chair* The suffix (~) expresses *nondeterministic*

choice. The agent *Advisor* does not know, for example, if the agents *Instructor* and *Chair* will reply to the sent messages.

4. Refined specification and protocol conformance

The protocol conformance can be verified during the design or implementation through an algorithm form, such as a state machine based comparison [11] [35]. This model defines an oracle to support the correctness verification in run-time. When an agent, in a certain state, receives a definitive event, it recognizes and evaluates what the event means, and then performs an action, resulting in a state transition. An action is an atomic unit of functionality that modifies the state of the system and the environment. When a protocol of an agent is carried out, a state transition diagram can be mapped by ECA rules (event-condition-action) [13] [24] [35]. The same action can be associated with more than one state transition. The previously shown *advIn* protocol (t_g) can be modeled as UML State Machine Diagram, as shown in Figure 3.

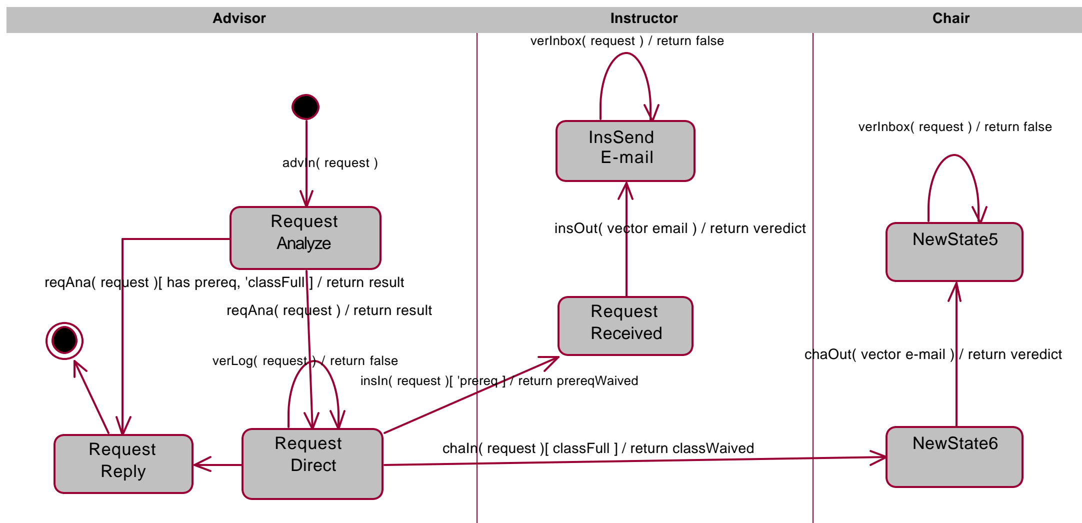

Figure 3 - The *advIn* protocol of the agent *Advisor* modeled as UML State Machine

The semantic of ECA rules is as follows. Whenever an ECA-event occurs, ECA-conditions are evaluated; if these conditions are evaluated as true, ECA-actions are performed. A set of ECA rules specify how an agent receives data from a specific type of message, performs local actions (e.g. to access database), sends messages and changes to a new state. The logical predicates restrict the set of valid states. The protocol contains nine states, partiality represented in Table 4:

States
State 1: initial state – Advsor receive request
State 2: Advisor analyze request
State 3: Advsor reply request
State 4: final state – return result
State 5: Advisor direct request
...

Table 4 - Partial states of the protocol *advIn*

Twelve transition rules exist in the protocol *advIn*, as partially shown in Table 5. After mapping the transitions for ECA rules, we have the following transitions rules in the behavior protocol:

Rules
Rule 1: State 1 → State 2
ON request
IF request 'null
THEN advisor.analizeRequest
Rule 2: State 2 → State 3
ON request data
IF ('classFull | prereq)
THEN result = 'ok'
...

Table 5 - Partial transition rules for the protocol *advIn*

Here, the problem of the state space can introduce an exponential growth, which can lead to a problem of states explosion. This problem can be solved factoring the state space [22] and performing tests for multiple abstraction levels, which results in substantial reduction of the considered state space. Lets consider MAS as a directed graph, where each node corresponds to an agent and the edges represent the communication between agents. For each set, the processing of the worse case is exponential in terms of number of used variables to test all the possible ways. However, we can limit the states set to be computed and tested, allowing only a small number of agents or modules to be tested in each increment. A substantial reduction of state space is then reached, as long as the total space is divided in lesser sets. The reduction of testing costs occurs by reducing the size and complexity of the test cases.

5. MAS evaluating

The formal specification of the systems leads to codes with less errors, and its correctness can be mathematically proved. The requirements specification is redefined in a detailed formal specification, which is expressed in a mathematical notation. However, the program proofs are long and costly, requiring expert work. Less orthodox processes such as Cleanroom [26] have been successfully applied in several domains, where rigorous processes of inspection and verification replace some of the mathematical proofs. In our approach, a specification less formal than the Cleanroom technique is utilized, and the system reliability is assured through the facilities provided by semi-automated inspections, facilities provides to automate test cases, applied together with an adequate testing strategy. In this section, we describe a testing strategy and techniques applied in our approach.

5.1. Testing strategy

When developing component-based MAS, the modules can be gradually integrated in order to produce the agent or the desired component. This integration must be processed step by step, assuring at the end of each step, the composed set of modules possesses good quality [31]. Each set corresponds to one *construct* [32]. A *construct* is a set of one or more devices forming a partial version, however operational, of the module or component. The process is carried out by developing and testing successive constructs, where each interaction is more comprehensive than the previous one. In our approach we applied a *mix-strategy* [32], using the best characteristics of the *top-down* and *bottom-up* strategies [9] [32] in order to extract the maximal advantage of each one.

First the agents are individually tested (agent abstraction); and later tested as a group (society abstraction). The programs must be developed in stages, where each stage produces the release of a new construct capable of being used for the purpose of evaluation. The modules or agents are incorporated in the construct as they are developed and tested. To be able to test the various modules as they are made, we must develop *test drivers* or command modules. The test drivers establish the necessary environment for testing. Normally, this module contains or is capable of activating several structures, exploring specs and functions of property measurements [32]. Moreover, the test drivers must possess user interfaces through which one can independently activate each one of the functions or methods of the modules under testing. To be able to test the root module without the use of correspondent server modules, it is necessary to develop *stubs*. These are modules of instrumentation capable of simulating server behavior modules not yet developed.

Testing of agents puts more emphasis on state-based testing [15] [23] than procedural code does. Determining the correctness of an agent involves determining whether it behaves properly in each of its possible states. There are two advantages [17] when applying the functional approach. First, functional testing is independent of how the software is implemented, that is, if the implementation changes, the test cases are still useful as long as the specification does not change. Secondly, the development test case can be performed in parallel with the implementation. Our main focus is the integration test. We consider that masses of unit testing have been applied and its results approved before the accomplishment of the integration tests. We also acknowledge the existence of some tools for unit testing in the market, such as JUnit [39] and Jtest [40] and many other approaches proposed in literature that address this question.

5.2. Verification and validation

Verification and validation (VV) [30] is the name given to the verification processes and analysis that assure that software fulfills its specifications and meets the requirements of the customers. The VV constitutes a process of complete life cycle , beginning with the revisions of the requirements and continuing with the revisions of design and inspections of code until arriving at the product tests. Inside of the VV process, we use in our approach two techniques for the checking and the analysis of systems:

- ✓ *software inspections*: analyze and verify the representations of the system, as the requirements document, design diagrams and the source code of the programs. The periods of training of the process can be applied in all development phases and can be complemented by some analysis in the text of origin of the system or associated documents. The automated inspections of software and analyses are static techniques of VV, and do not require the system execution. The inspection techniques include automated source code inspections and formal verification.
- ✓ *software testing*: involves the execution of an implementation of the software with the test data and to examine the output results and its operational behavior, in order to verify if it is being executed in agreement with the expected results. The tests are a dynamic technique of verification and validation because they work with an executable representation of the system.

Test cases are implemented using test scripts that are a necessary part of the test automation. Scripts express the actions that would be executed by the tester, such as: (i) establish the sequences of execution; (ii) inform the input data and (iii) compare obtained with expected

results. In the case of an agent, we force this agent to be in a predetermined state, and then exercise its behavior for different situations. Equivalence partition sets [5] [9] [10] are then selected for input data in order to exercise the agent's behavior. The techniques to produce scripts are similar to the programming techniques. The test script skeleton contains the code to perform the test sets that compose the test case. Test case execution can be automated using a tool that can read and interpret this script skeleton.

The *Testing Framework* (under construction) can be defined such as a set of cooperating classes for instantiating the testing artifacts necessary to perform and manage the VV process. Figure 4 shows the various models that make-up the test environment. *Test Model* is composed by two modules: *Test Generator* and *Test Manager*.

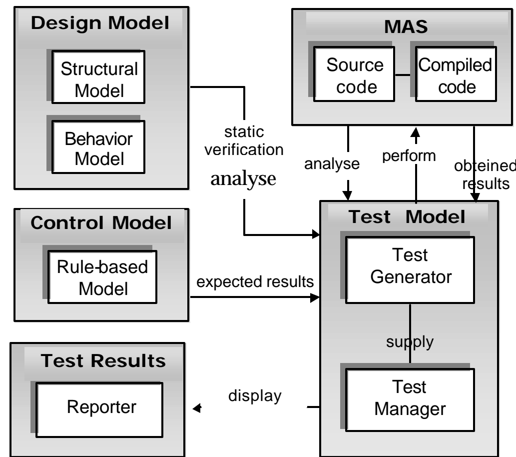

Figure 4 – The testing environment

Test Generator provides a set of classes that allow the instantiating of test scripts and test cases to perform automated state-based testing. In order to carry out state-based testing, we need a means to test and set a desired state of the agent, in order to observe its behavior and so verify the agent and MAS behavior compliance. The conformance of the design representation with the implemented code is done by automated inspections, which can be done by instancing extending abstract classes. *Test Manager* provides a set of graphical interfaces for management, configuration and execution of test inspections and test cases. The purpose is to aid the tester in constructing and select test processes by use of manual and automated fashion, as well as repeatedly adapting and rerunning those tests for regression testing purposes.

The *Control Model* is the base of state-based testing; it defines an oracle for the protocols and can be generated after the refined specifications have been made as described in Section 4. The general principle is to test if each transition of the specification reaches the destination state, applying the input and checking that the output is correct and verifying the target state. Control model stores protocols and rules, with the invariants represented by lists of predicates. The objective here is to consider the specification and the implementation as state machine in which transitions are labeled with inputs and outputs. The control model can supports automatic verification of valid states and stores the rules that define the correctness of protocols in all points of the process and the results that must be expected and verified during the test.

6. Related works

The reuse of generic software abstractions is recognized within object-oriented and components development community and has lead to the concepts such as frameworks and design patterns. Many of the principles here described for agent components are a result of previous research [4] applied for components' development. Although the models and abstractions are for different domains, and consequently for different composites of structures, micro-frameworks and behavior model, in both cases there is potentialization of reuse.

Frameworks for multi-agent systems can be found in [36] [37] [38]. AUML extends UML with enhanced interaction diagrams to make more explicit some of the message and protocol handling [14]. The FIPA 2000 framework provides a definition of an abstract architecture, allowing alternative implementations that will interoperate. FIPA-OS (FIPA Open Source) [38] is an open-agent platform that supports communication by using the FIPA agent comunication language standards. The latest open-source code is found at http://fipa-os.sourceforge.net/.

Zeus [37] is a 'collaborative' agent building environment and component library written in Java. Each ZEUS agent consists of a definition layer, an organizational layer and a co-ordination layer. The definition layer represents the agent's reasoning and learning abilities, its goals, resources, skills, beliefs, preferences, etc. The organization layer describes the agent's relationships with other agents. The co-ordination layer describes the co-ordination and negotiation techniques the agent possesses. Communication protocols are built on top of the co-ordination layer and implement inter-agent communication. Beneath the definition layer is the API.

JADE (Java Agent DEvelopment Framework) [38] is a software framework to develop agent-based applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. The goal is to simplify the development while ensuring standard compliance through a comprehensive set of system services and agents. JADE can be considered an agent middle-ware that implements an agent platform and a development framework. It deals with all those aspects that are not peculiar of the agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle. All agent communication is performed through message passing, where FIPA ACL is the language to represent messages.

7. Conclusion and ongoing works

Our main contribution is the investigation, integration and adaptation of a set of software engineering techniques, such as components architecture, formal behavior representation, rule-based systems, design patterns and testing techniques for building a framework to support multi-agent systems development. Our framework and the concepts presented in this paper can be used to a rapid and large-scale development of flexibile, reliable and reusable MAS. The reuse is achieved by architectural model propagation for multiple components and by the use of design patterns foundations, such as encapsulation, high-cohesion and low-coupling. Support for dynamic compositions is achieved via mediators, and they guarantee highly flexible and extensible structures.

Although the formal specification used in our approach might help to produce safer and more reliable systems, we are aware that it does not guarantee software reliability. However, the specification model proposed, as well as the techniques of verification and validation applied at design-time and run-time warrant high quality and reliability. The model

facilitates the testing process, so that specifications can be easily retrieved when generating test scripts skeletons. The inspections are widely facilitated by the use of automated tools, and additional information contained in specification and control model, such as expected results, pre/post-conditions and transition rules. In that way, the adherence to the requirements can be warranted by execution of significant test cases for behavior agent testing in run-time.

In addition, our research work will address the following questions:

- Following the first phase of the work, our focus will be directed to improve the implementation of framework for testing execution and management. We are now interested in simplifying the task of testing requirement specification, while improving the automatic generation of test case skeleton scripts.
- The testing of concurrent programs [19] [21] [27] [46] is difficult due to the inherent non-determinism in these programs. That is, if we run a concurrent test twice with the same test input, it is not guaranteed that the resulting output will be the same for both cases. This non-determinism causes two significant test automation problems [46]: (i) it is hard to force the execution of a given program statement or branch, and (ii) it is difficult to automate the checking of test outputs. It requires the queuing of messages and events, in such way that buffers of communication need to be included. In this case, the complexity can quickly raise scalability problems that need to be analyzed.
- Software testing is usually performed after the production of code. However, it has been observed that the later the error is detected, the more expensive is the correction [5]. This comment encourages the investigation of techniques that apply the test before software development. TfD (Test-first-Design) is one of the XP (Extreme Programming) [1] [17] most important practices; it requires that any production of code should not be implemented before writing the unit testing. Here, our proposal is that this technique could be extended in order to require that integration test cases between agents could also be designed before building the definitive code.
- To support reliability measure, we are developing a statistical database. In order to measure the reliability of MAS, we can periodically consult the statistical database, showing the variance of reliability improvement.
- We are also developing a FIPA compliant protocol translator to connect different agent platforms messages for interoperable intelligent multi-agent systems. When the sender or receiver does not belong to the same platform, the message is automatically converted to/from the FIPA compliant string format. In this way, this conversion is hidden from the agent implementers that only need to deal with the same class or Java object.

Acknowledgment

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

References

[1] **Beck, K.** *Extreme Programming Explained: Embrace Change*; Massachusetts: Addison Wesley Longman; 1999.

[2] **Bigus J.** et al. "Constructing Intelligent Agents Using Java". John Wiley & Sons, Inc. New York, 2001.

[3] **Cheikhhrouhou M.M.**, **Lebetoulle J**. "When Agents Become Autonomous, How to Ensure their Reliability?" Technical Report. *Institut Eurecom, Corporate Communications Dep.*, France, 1999.

[4] **Caminada N., Haendchen Filho A., Godoy R., Staa A.v.** "A Model for Component Development and Reuse Applying Frameworks and Design Patterns." *Proceedings of 3rd. Workshop on Component-Based Development*. Universidade Federal de São Carlos, São Paulo, Brasil, 2003.

[5] **DeMillo R.A. et al.** "Software Testing and Evaluation." Benjamin/Cummings Publishing Company, Inc. Menlo Park, California, 1987.

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

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

[8] **Ferber et al.** "A Meta-model for the Analysis and Design of Organizations in Multi-Agent Systems". *Proceedings of ICMAS*, Paris, 1998.

[9] **Fewster M. et al.** "Software Test Automation: Effective Use of Test Execution Tools". Addison Wesley, New York, 1999.

[10] **Frankl P.G. et al.** "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing". *IEEE Transactions On Software Engineering* August 1993.

[11] **Fujiwara S**. "Test Selection Based on Finite State Models". *IEEE Transactions on Software Engineering* 1991.

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

[13] **Gelfond M**. "Representing Action and Change by Logic Programs". *The Journal of Logic Programming*. Elsevier Science Publishing Co, New York 1993.

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

[15] **Hartmann J., Imoberdorf C., Meisinger M.** "UML-Based Integration Testing". *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. Portland, Oregon, USA, 2000.

[16] **Haendchen Filho A., Staa A.v., Lucena C.J.P.** "A conceptual role-based model for building and managing multi-agent software testing". *Proceedings of SELMAS-2002 - Software Engineering for Large-Scale Multi-Agent Systems*, in conjuntion with ICSE-2002. Orlando, Florida, USA, May 2002.

[17] **Jeffries R**. "Extreme Programming Installed". Addison-Wesley, Boston, 2001.

[18] **Jennings N**., **Sycara K**. and **Wooldridge M**. "A Roadmap of Agent Research and Development". *Autonomous Agents and Multi-Agent Systems*, 1(1), 1998.

[19] **Khoumsi A**. "A Temporal Approach for Testing Distributed Systems". *IEEE Transactions on Software Engineering*, vol. 28, N. 11, Nov. 2002.

[20] **Larman C.** ‘Applying UML and Patterns”. Prentice Hall PTR. Upper Saddle River, NJ, USA, 1998.

[21] **Milner R**. “A Calculus of Communicating Systems”. Springer Verlag, 1980.

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

[23] **Plasil F. et al.** “SOFA/DCUP - Architecture for Component Trading and Dynamic Updating”. *Proceedings Fourth Conference Configurable Distributed Systems* (ICCDS’98), 1998.

[24] **Paton N.W**. “Supporting Production Rules Using ECA-Rules in an Object-Oriented Context”. *Department of Computer Science.* Technical Report. *University of Manchester*, Manchester, UK, 1995.

[25] **Potter B**., **Sinclair J**., **Till D**. “Introduction to Formal Specification in Z”. Prentice Hall PTR, 1997.

[26] **Prowell S.J., Trammel C.J.** “Cleanroom software engineering: technology and process”. Addison-Wesley Longman, 1999.

[27] **Roscoe A.W**. “The Theory and Practice of Concurrency”. Prentice Hall, 1998.

[28] **Shaw M., Garlang D.** “ Software architecture: perspectives on an emerging discipline.” Prentice Hall, 1996.

[29] **SOFA Project**. http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html, 2000.

[30] **Sommerville I**. “Software Engineering. Pearson Education Limited”. Addison Wesley, England, 2001.

[31] **Souter A. et al.** “OMEN: A Strategy for Testing Object-Oriented Software”. *Proceedings of the ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*. Portland, Oregon, USA, 2000.

[32] **Staa A. v**. “Modular Programming”. Editora Campus Ltda. Rio de Janeiro, Brasil, 2000.

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

[34] **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.

[35] **Yu L. et al.** “A Conceptual Framework for Agent Oriented and Role Based Workflow Modeling.” Technical report, *Institute for Media and Communications Management*, University of St. Gallen, Switzerland, 2000.

[36] **Azarmi N., Thompson S.** ZEUS: A Toolkit for Building Multi-Agent Systems. Proceedings of *fifth annual Embracing Complexity Conference*, Paris April 2000.

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

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

[39] **PARASOFT**. “JTest: Automatic Java Software and Component Testing”. Monrovia, CA, USA, 2002.

[40] **Beck**, **K**. “Testing Resources for Extreme Programming”. http://www.junit.org /index.htm.

[41] **Spivey J.M.** “The Z Notation: a Reference Manual”. Prentice-Hall, Londres, 1999.

[42] **Wordsworth J.** “Software Engineering with B”. Addison Wesley, New York, 1996.

[43] **Jones C.B.** “Systematic Software Development Using VDM”. Prentice-Hall, Londres, 1986.

[44] **Hoare C.A.R.** “Communicating Sequential Processes. Prentice-Hall, Londres, 1985.

[45] **Peterson J.L.** “Petri Net Theory and the Modeling of Systems. McGraw-Hill, New York, 1981.

[46] **Long B. et al.** “Tool Support for Testing Concurrent Java Components”. IEEE Transactions on Software Engineering, vol. 29, N. 6, June 2003.

[47] **Guttag J. et al.** “The Larch family of specifications languages”. IEEE Software, 2(5), p. 24-36, 1985.