A Formal Development and Validation Methodology Applied to Agent-Based Systems

Di Marzo Serugendo, Giovanna

A Formal Development and Validation Methodology Applied to Agent-Based Systems [⋆]

Giovanna Di Marzo Serugendo

CERN/IT Division
CH-1211 Geneva 23, Switzerland
Giovanna.Di.Marzo@cern.ch
http://www.cern.ch/Giovanna.Di.Marzo

Abstract. This paper presents first a formal development methodology that enables a specifier to add complexity progressively into the system design, and to formally validate each step wrt client's requirements. Second, the paper describes the application of this methodology to agent-based systems, as well as development guidelines that help the specifier during the development of such systems. The methodology and the development guidelines are presented through an agent market place example.

1 Introduction

Multi-agent systems need, as any other system, to be supported by a proper development methodology. The need for such a methodology is more crucial in the case of agent-based systems, since the composition of independently developed agents may lead to unexpected emergent behaviour. In addition, agent-based systems are complex, and it is difficult for a specifier or a programmer to put every details immediately into his design.

This paper presents a *formal development methodology* that enables the designer to add complexity progressively into the system: problems are solved one after the other, and design decisions are formally validated at each step. The methodology follows the two languages framework, i.e., it advocates the joint use of a model-oriented specifications language for expressing the system's behaviour, and a property-oriented specifications language (logical language) for expressing properties. The proposed methodology is general enough and can be applied to any model-oriented formal specifications language. A particular application has been realised for a special kind of synchronized Petri nets, called CO-OPN/2 [1].

This paper describes as well *development guidelines for agent-based systems* within the proposed methodology. Agent decomposition, interactions between agents (composition, coordination, message passing, blackboard), as well as implementation constraints (e.g., actual communication using RMI, CORBA, etc.) are progressively added during the development process.

[⋆] Part of this work has been performed while the author was working at the University of Geneva and at the Swiss Federal Institute of Technology in Lausanne (EPFL).

The structure of this paper is as follows. Section 2 describes the formal development methodology, and presents the formal specifications language CO-OPN/2. Section 3 provides the development guidelines for agent-based systems. Section 4 illustrates the methodology and guidelines through a simple agent market place example. Section 5 presents related works.

2 Development Methodology

The proposed methodology addresses the three classical phases of the development process of distributed applications: the analysis phase, the design phase, and the implementation phase. This section presents the design phase, explains the necessity of development guidelines, and briefly describes CO-OPN/2.

2.1 Design by Contracts

The analysis phase produces informal requirements that the system has to meet. The design phase consists of the stepwise refinement of model-oriented specifications. Such specifications explicitly define the behaviour of a system, and implicitly define a set of properties (corresponding to the behaviour defined by the specification). During a refinement step it is not always necessary, desirable or possible, to preserve the whole behaviour proposed by the specification. Therefore, essential properties expected by the system are explicitly expressed by means of a set of logical formulae, called *contract*. A contract does *not* reflect the whole behaviour of the system, it reflects only the behaviour part that must be preserved during all subsequent refinement steps. A refinement is then defined as the replacement of an abstract specification by a more concrete one, which respects the contract of the abstract specification, and takes into account additional requirements.

The implementation phase is treated in a similar way as the design phase. At the end of the design phase, a concrete model-oriented specification is reached, it is implemented, and the obtained program is considered to be a correct implementation if it preserves the contract of the most concrete specification.

Figure 1 shows the three phases. On the basis of the informal requirements, an abstract specification Spec_0 is devised. Its contract $\mathrm{Contract}_0$ formally expresses the requirements. During the design phase, several refinement steps are performed, leading to a concrete specification Spec_n and its contract $\mathrm{Contract}_n$. The implementation phase then provides the program Program and its contract Contract. A refinement step is correct if the concrete contract contains the abstract contract.

This methodology is founded on a general theory defined in [4]. The particularity of this methodology wrt traditional ones using the two languages framework is that it goes a step further, since the contracts explicitly point out the essential properties to be verified. Indeed, the specifier can freely refine the formal specifications, without being obliged to keep all the behaviour.

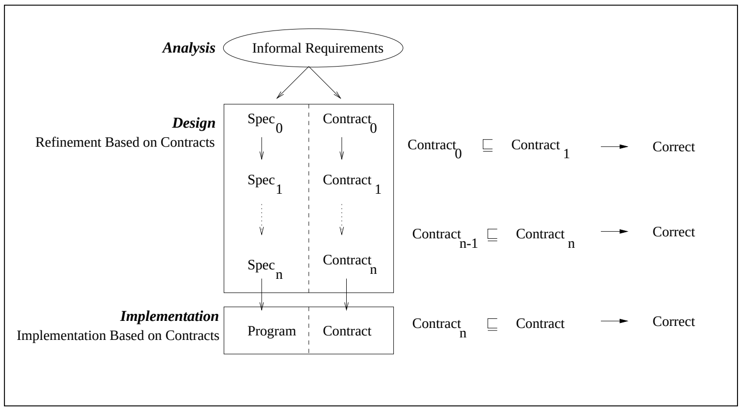
Analysis
Informal Requirements
Design
Refinement Based on Contracts
Spec0
Contract0
Spec1
Contract1
Specn
Contractn
Contract0 ⊑ Contract1 → Correct
Contractn-1 ⊑ Contractn → Correct
Implementation
Implementation Based on Contracts
Program
Contract
Contractn ⊑ Contract → Correct

Fig. 1. Development Methodology

This methodology is well-suited for agent-based systems, since complexity is introduced progressively, and emergent behaviour can be controlled by the means of contract.

2.2 Development Guidelines

The theory of refinement and implementation based on contracts provides the basis to formally prove that a refinement step and the implementation phase are correct. However, the theory cannot help the specifier in establishing a contract, and in choosing a more concrete specification. Therefore, we suggest the use of *development guidelines*, i.e., a sequence of refinement steps that a specifier should follow when developing an application. Development guidelines depend on the kind of application being developed. They should be seen as refinement patterns, since, after having identified the system to develop, the specifier applies a dedicated series of design steps.

Development guidelines have already been defined for client/server applications [3], as well as for dependable applications [5].

2.3 CO-OPN/2 and HML

The above general theory has been applied to a high-level class of Petri nets, called CO-OPN/2, using the Hennessy-Milner logic (HML) as logical language. Examples of this paper will be illustrated using CO-OPN/2 and HML.

CO-OPN/2 [1] is an object-oriented formal specifications language. An object is defined as an encapsulated algebraic net in which places compose the internal state and transitions model the concurrent events of the object. Transitions are either methods (callable from other objects), or internal transitions (describing the internal behaviour of an object). Objects can be dynamically created. Each object has an identity that can be used as a reference. When an object requires a service, it asks to be synchronised with the method of the object provider (`with`). The synchronisation policy is expressed by means of a synchronisation expression, which can involve many partners joined by three synchronisation operators (simultaneity (`//`), sequence (`..`), and alternative (`+`)).

An HML formula, expressed on CO-OPN/2 specifications is a sequence (or a conjunction ($\wedge$), or an alternative ($+$)) of observable events (firing of a single method or parallel firing of several methods). An HML formula is satisfied by the model of a CO-OPN/2 specification if the sequence of events defined by the formula corresponds to a possible sequence of events of the model of the specification.

3 Agent-Based Systems

There is currently no general consensus on the definition of an agent. Therefore, this section first presents some preliminary definitions of what we think are an agent, and an agent-based system. Second, it describes the development guidelines identified for these systems.

3.1 Definitions

From a software engineering point of view, we consider an agent-based system in the following manner:

- the system performs some *functionality* to some final user (another software system, human being, etc.);
- the system is made of one or more *collections* of agents, together with *relationships* among collections (negotiation techniques, cooperation protocols, coordination models). Agents engage in collections, that can change at run-time (joint intentions, teams);
- agents in a collection *interact together* to solve a certain goal (message passing, blackboard, etc). They may have some social knowledge about their dependencies (peers, competitors);
- an agent is a *problem-solving entity.* It performs a given algorithm to reach its goal.

3.2 Development Guidelines

The development steps identified in the case of agent-based applications are the following:

1. *Informal Requirements:* a set of informal application's requirements including validation objectives is defined;
2. *Initial contractual specification: System's functionality.*
   Based on the informal requirements, the initial specification provides an abstract view of the system where the problem is *not* agent based. The contract reflects the functionality of the application;

3. *First refinement step: System's collections.* This step leads to a view of the system made of several collections of agents, together with the relationship among the collections (e.g. joint intentions, teams, etc.). The contract is extended to the functionality of each collection, and to the properties of their composition;
4. *Second refinement step: Collections design.* Each collection is specified as a set of agents together with their interactions (message passing, blackboard, dependencies). The contract describes the functionality of each agent in the collection, as well as the desired properties of the agents interactions;
5. *Third refinement step: Agent design.* The internal behaviour of each agent is fully described (algorithm used for solving its goal, action decision upon knowledge processing, etc.). The contract is extended to the properties expected by the internal behaviour of each agent;
6. *Fourth refinement step: Actual communications means.* The previous steps define the high-level communication means employed by the agents. This step integrates the low-level communications means upon which high-level communications can be realised (RMI, CORBA, etc.). The contract contains the characteristics of the chosen communication;
7. *Implementation.* Step 6 is implemented using the chosen programming language. The contract of step 6 is expressed on the program.

These guidelines enable the macro-level part (identification of collections of agents) to be followed by a micro-level part (design of collections, design of agents). In addition, the micro-level part can be done independently for every collection (steps 4. to 7.).

4 Market Place Example

We consider a simple market place example based on [2]. This section describes the development of this system according to the guidelines given above.

4.1 Informal Requirements

This step corresponds to guideline 1. The market place application offers some operations to buyers and sellers that have to respect the following requirements:

- A new buyer can register itself at any moment to the market place system;
- A new seller can register itself at any moment to the market place system;
- A registered buyer can propose a price for a given item that he wants to buy, and specifies the highest price that he is ready to pay for the item;
- A registered seller can make an offer for a given item that he wants to sell, and specifies the lowest price at which he is ready to sell the item;
- Buyers and sellers can consult the system to know if they have been involved in a transaction. The price reached during the transaction must be less or equal to the highest price specified by the buyer, and greater or equal to the lowest price specified by the seller.

4.2 Initial Specification: Functional View

The initial specification corresponds to development guideline 2. It is given by CO-OPN/2 specification made of `MarketPlace` class of Fig. 2.

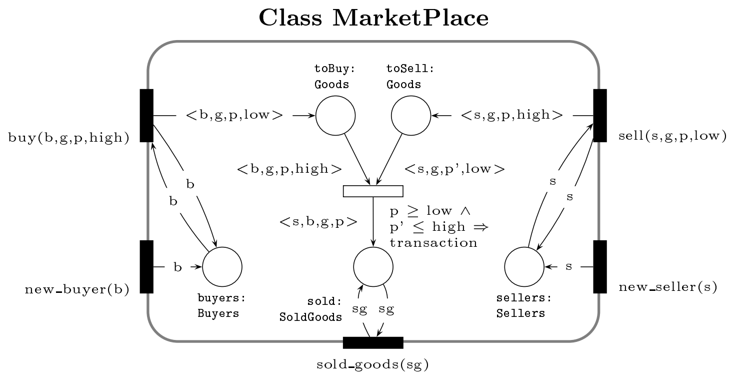

Fig. 2. Market Place System

This class offers five methods, corresponding to the five system operations identified in the previous design step (Section 4.1):

- the `new_buyer(b)` method is used by a buyer whose identity is `b` for registering itself to the system. The system simply enters identity `b` into place `buyers`;
- the `new_seller(s)` method is used by a seller, called `s`, for registration. The system simply enters identity `s` into place `sellers`;
- the `buy(b,g,p,high)` method enables an already registered buyer `b` to inform the system that he wants to buy an item `g` at a desired price `p`. The highest price he is ready to pay for the item is `high`. The system then enters this information into place `toBuy`;
- the `sell(s,g,p,low)` method enables an already registered seller `s` to propose the item `g`, with a starting price `p`, and a minimum price `low`. The offer is entered in place `toSell`.
  As soon as there is a request for buying item `g` and an offer concerning the same item `g`, with a buying price compatible with the lowest selling price, and a selling price compatible with the highest buying price, the transaction occurs. The request for buying and the offer are removed from the system by transition `transaction`, and the transaction is entered into place `sold`;
- the `sold_goods(sg)` method enables a buyer (or seller) to consult the system for occurred transactions.

Contract. In order to remain concise, we present a contract $\phi_{\mathbf{I}}$, expressed on this initial specification, made of only two HML formulae: $\phi_{\mathbf{I}_1}$ and $\phi_{\mathbf{I}_2}$. It is obvious that a larger contract is necessary to ensure all the informal requirements.

Assuming variables such that $l \leq p1 \leq h$, $p2 \leq l$ and $p3 \geq h$:

$$\begin{aligned}
\phi_{\mathbf{I}_1} = & <MP.\,\text{create}><MP.\,\text{new\_buyer}(b)><MP.\,\text{new\_seller}(s)> \\
& <MP.\,\text{buy}(b, g, p, h)><MP.\,\text{sell}(s, g, p', l)><MP.\,\text{sold\_goods}(s, b, g, p1)> \\
\phi_{\mathbf{I}_2} = & <MP.\,\text{create}><MP.\,\text{new\_buyer}(b)><MP.\,\text{new\_seller}(s)> \\
& <MP.\,\text{buy}(b, g, p, h)><MP.\,\text{sell}(s, g, p', l)> \\
& (\neg <MP.\,\text{sold\_goods}(s, b, g, p2)> \wedge \neg <MP.\,\text{sold\_goods}(s, b, g, p3)>)\ .
\end{aligned}$$

Formula $\phi_{\mathbf{I}_1}$ states that once the market place MP has been created, a buyer b, and a seller s can register themselves to the system. They can respectively make a request to buy item g, and an offer to sell item g. Then, the transaction occurs for prices $p1$ compatible with the lowest selling price, and with the highest buying price, i.e., such that $l \leq p1 \leq h$.

Formula $\phi_{\mathbf{I}_2}$ is similar to $\phi_{\mathbf{I}_1}$, but it states that for prices $p2$ such that $p2 \leq l$ and prices $p3$ such that $p3 \geq h$, then the transaction does *not* occur.

Contract $\phi_{\mathbf{I}}$ is actually satisfied by the model of the initial specification. Indeed, transition `transaction` is guarded by condition $p \geq low \wedge p' \leq high$. This condition prevents the firing of this transition whenever it does not evaluate to true.

4.3 Refinement R1: Agent Decomposition and Interactions

This step corresponds to development guideline 4. (In this example, step 3. is skipped, because the system contains only one collection of agents.)

The specification is made of three classes: the `MarketPlace` class, given by Fig. 3, the `BuyerAgents` class of Fig. 4, and the `SellerAgents` class of Fig. 5.

The `MarketPlace` class stands for the homonymous class of the initial specification. It offers the same interface as before to the actual buyers and sellers, enriched with some more methods:

– the `new_buyer(b)` and `new_seller(s)` methods enable a new buyer `b`, or a new seller `s` to enter the system. A dedicated agent `b_agent`, respectively `s_agent` is created. The system stores pairs, made of a buyer's identity and the identity of its dedicated agent, into place `buyers`; and pairs of seller's identity and agent's identity into place `sellers`;
– the `buy(b,g,p,high)` and `sell(s,g,p,low)` methods are used by buyer `b`, respectively seller `s`, to enter a request to buy an item, respectively an offer to sell an item into the system. The market place forwards this information to the agent that works on behalf of the buyer or the seller. It retrieves the identity of the corresponding agent, and calls the method `new_good(g,p,high)`, respectively `new_good(g,p,low)`;
– the `sold_goods(sg)` method enables buyers and sellers to consult the list of transactions;
– the `get_buyers(l)` and `get_sellers(l)` methods return the list of all buyer agents, and seller agents respectively. Method `get_buyers(l)` is used by seller agents to know the identities of buyer agents, in order to ask them

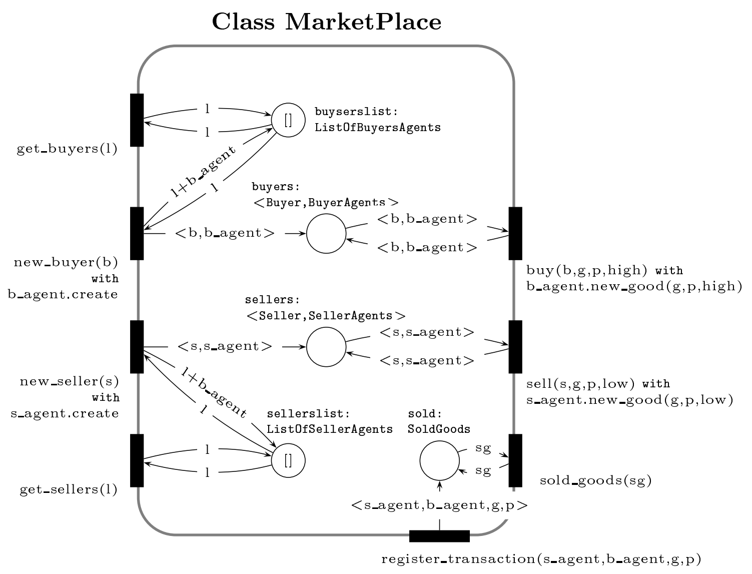

Fig. 3. Refinement R1: Market Place System

some services. Similarly, method `get_sellers(l)` is used by buyer agents to know identities of seller agents;

– the `register_transaction(s_agent,b_agent,g,p)` method is used by seller or buyer agents to inform the system about the transactions that have occurred.

The `BuyerAgents` class, given by Fig. 4, specifies buyer agents, while the `SellerAgents` class, given by Fig. 5, specifies seller agents. These classes are very similar, and behave almost in the same manner.

– the `create` constructor of the `BuyerAgents` class enables to create new instances of buyer agents;
– the `new_good(g,p,high)` method is called by the market place whenever the buyer (for whom the agent is working) enters a request to buy an item into the system. The agent stores the request into place `toBuy`. As soon as the request is stored in this place, transition `makeOffers` first contacts the market place in order to obtain the current list of sellers (this list changes when the system evolves, since new sellers can enter the system at any moment). Second, the transition informs every seller of this list (broadcast) that there is a new request for buying item `g`, by calling method `sendOffer` of each seller agent. If after some time, no transaction concerning this request has occurred, transition `timeout` increases the price from one unit (provided that the highest price condition is not violated);
– the `sendOffer(s_agent,g,p)` method is used by a seller agent, whose identity is `s_agent`, to inform the buyer agents that it sells item `g` at price `p`. As soon as the there is an offer for item `g` at a price `p`, which is the same as the

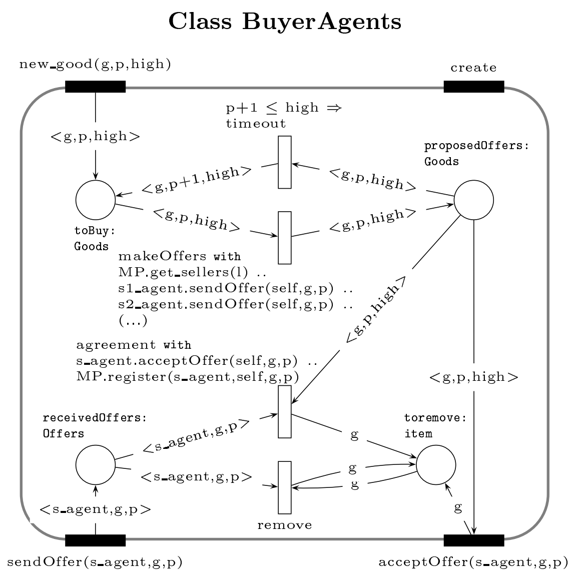

Fig. 4. Refinement R1: Buyer Agent

current price offered by the buyer agent, the transaction occurs. Transition `agreement` fires: it calls method `acceptOffer` of the corresponding seller agent (`s_agent`), and informs the market place. Due to the CO-OPN/2 semantics, transition `agreement` can fire only if method `acceptOffer` of the corresponding seller agent can fire. In that manner, only one agreement can be reached for a given offer (the seller does not sell two times the same item). Indeed, if the seller agent has already reached an agreement with another buyer, then its method `acceptOffer` cannot fire, and consequently transition `agreement` of the current buyer cannot fire;

– the `acceptOffer(s_agent,g,p)` method is called by a seller agent, whose identity is `s_agent`, when an agreement is reached with the buyer agent.

The `SellerAgents` class is similar to the `BuyerAgents` class, except that these agents decrease their prices when there is no corresponding buyer.

Contract. The contract for refinement R1 is made of three formulae. Considering, as before, variables such that: $l \leq p1 \leq h$, $p2 \leq l$ and $p3 \geq h$, the contract is made of the three formulae below:

$$\phi_{\mathbf{R1}_1} = \phi_{\mathbf{I}_1}, \phi_{\mathbf{R1}_2} = \phi_{\mathbf{I}_2}$$

$$\begin{aligned}\phi_{\mathbf{R1}_3} = {} & <MP.\,\text{create}><s\_agent.\,\text{create}><b1\_agent.\,\text{create}><b2\_agent.\,\text{create}> \\ & <s\_agent.\,\text{sendOffer}(b1\_agent, g, p)><s\_agent.\,\text{sendOffer}(b2\_agent, g, p)> \\ & <b1\_agent.\,\text{sendOffer}(s\_agent, g, p)><b2\_agent.\,\text{sendOffer}(s\_agent, g, p)> \\ & ((<b1\_agent.\,\text{acceptOffer}(s\_agent, g, p)> + <b2\_agent.\,\text{acceptOffer}(s\_agent, g, p)>) \wedge \\ & \neg\, (<b1\_agent.\,\text{acceptOffer}(s\_agent, g, p)><b2\_agent.\,\text{acceptOffer}(s\_agent, g, p)>) \wedge \\ & \neg\, (<b1\_agent.\,\text{acceptOffer}(s\_agent, g, p)> \;//\; <b2\_agent.\,\text{acceptOffer}(s\_agent, g, p)>))\ .\end{aligned}$$

Formulae $\phi_{\mathbf{R1}_1}$, and $\phi_{\mathbf{R1}_2}$ are the same as $\phi_{\mathbf{I}_1}$ and $\phi_{\mathbf{I}_2}$. Formula $\phi_{\mathbf{R1}_3}$ states that once the market place has been created, it is possible to create a seller agent *s_agent*, and two buyer agents *b*1*_agent* and *b*2*_agent*. The seller agent offers item g at price p, and the two buyer agents are ready to pay the same price p for g. The formula then states that either buyer agent *b*1*_agent* or *b*2*_agent* accepts the offer (`+`), but *not* both (neither in sequence, nor simultaneously (`//`)).

The three formulae of the contract are satisfied by the specification. Indeed, formulae $\phi_{\mathbf{R1}_1}$ and $\phi_{\mathbf{R1}_2}$ are true, because of the guarded transition `timeout`. There is no request (nor offer) that violates the condition $p+1 \leq high$ (respectively $p-1 \geq low$).

Formula $\phi_{\mathbf{R1}_3}$ is true because transition `agreement` can fire only once per transaction: either transition `agreement` of the buyer agent fires, or that of the seller agent fires, but not both. The firing of transition `agreement` requires the firing of the method `acceptOffer` of the other agent involved in the transaction. The firing of these methods causes the removal of token <`g,p,high`> from place `proposedOffers` of the buyer agent, and <`g,p,low`> from place `proposedOffers` of the seller agent. In that manner, transition `agreement` and method `acceptOffer` cannot fire more than once for each offer.

Although the internal behaviour of the initial specification and the first refinement are different (the agreement is reached on a different basis), the specification of the first refinement is actually a correct refinement of the initial specification. Indeed, the contract of the initial specification is preserved by the refinement R1.

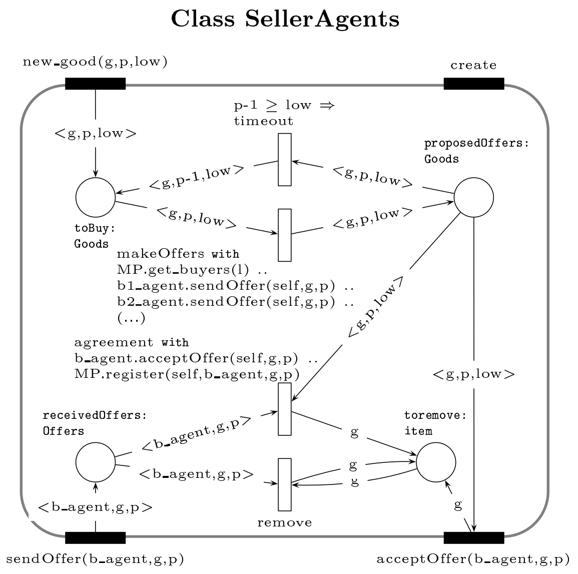

Fig. 5. Refinement R1: Seller Agent

4.4 Refinement R2: Actual Communications

This step corresponds to development guideline 6. (Step 5 is skipped, since, in this example, we do not want a more sophisticated agent algorithm).

In the case of an electronic market place, communications among agents occur through the Internet. Therefore, mechanisms such as RMI, CORBA, sockets, etc. have to be considered, and chosen.

In the case of our example, an RMI mechanism (based on TCP/IP) has been considered. The market place acts as a server: it provides some RMI object to the agents so that they access the market place through this object as if it was local. Agents are specified as RMI objects, thus remote invocation may occur from the market place to the agents and between agents. The specification is made of 5 classes: the market place (acting as a server); an RMI class for accessing the market place; two RMI classes for the buyer agents, and the seller agents respectively; and an additional class representing the RMI registry.

Contract. The contract of section 4.3 is extended to take into account RMI features.

4.5 Implementation

This step corresponds to development guideline 7. A Java program is derived from the previous step. Each CO-OPN/2 class is implemented in Java.

Contract. The contract contains the same formulae as the contract of the previous step, but expressed on the Java program, instead of the CO-OPN/2 specification, e.g., the creation of the system is represented by the call to the `main` method of the program (Java Class `MarketPlace`). Description of such translation is given in [4].

5 Related Works

Agent-oriented software engineering is currently a subject of increasing research. Jennings [6] describes agent-based systems under a software engineering point of view: agents, high-level interactions, and organisational relationships. Gaia [8] is a methodology defined for agent-oriented analysis and design. It enables to develop a system increasingly. The specifier describes the system using several models: requirements, roles models, interactions models (for the analysis); and agent model, services model, acquaintance model (for the design).

The verification that a program is correct wrt system specifications is a problem similar to the one of verifying that system specifications are correct wrt the requirement specifications. Meyer [7] advocates that, in order to face the problem of correctness, every program operation (instruction or routine body) should be systematically accompanied by a pre- and a post-condition.

6 Summary

This paper presents a methodology for developing agent-based systems that enables progressive system design and formal validation of each step. The paper presents as well development guidelines, that help the specifier to introduce complexity into the design. A small agent market place system is described: starting from informal requirements a Java implementation is reached, and every step is formally proved.

References

1. O. Biberstein, D. Buchs, and N. Guelfi. CO-OPN/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Chapman and Hall, 1997.
2. A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1996.
3. G. Di Marzo Serugendo. A formal development and validation methodology for system design. In *5th International Conference on Information Systems Analysis and Synthesis (ISAS'99)*, 1999.
4. G. Di Marzo Serugendo. *Stepwise Refinement of Formal Specifications Based on Logical Formulae: from CO-OPN/2 Specifications to Java Programs.* PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1999.
5. G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. F. Zorzo. Formal development and validation of Java dependable distributed systems. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99).* IEEE Computer Society Press, 1999.
6. N. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2000):277–296, 2000.
7. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.
8. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(2000), 2000.