Specification of Role-based Interactions Components in Multi-Agent Systems

Nabil Hameurlain[1], Christophe Sibertin-Blanc[2]

[1] *LIUPPA Laboratory, Avenue de l'Université, BP 1155, 64013 Pau, France*
nabil.hameurlain@univ-pau.fr
http://www.univ-pau.fr/~hameur
[2] *IRIT Laboratory, University Toulouse I, Place Anatole France31042 Toulouse, France*
sibertin@univ-tlse1.fr

Abstract. Roles are an important concept used for different purposes as the modeling of the organizational structure of multi-agent systems, the modeling of protocols, and as basic building blocks for defining the behavior of agents. Modeling interactions by roles brings several advantages, the most important of which is the separation of concerns by distinguishing the agent-level and system-level with regard to interaction. However, in open MASs, the composition of independently developed roles can lead to unexpected emergent interaction among agents. This paper identifies requirements for modeling role-based interactions, and presents a formal specification model of roles for complex interactions. Our approach aims to integrate specification and verification of roles into the Component Based Development approach. An interaction protocol example is given to illustrate our formal framework.

1 Introduction

The Multi-Agent System (MAS) paradigm is one of the most promising approaches to create open and dynamic systems, where heterogeneous entities are naturally represented as interacting autonomous agents, which can enter or leave the system at will. Interaction among autonomous agents is fundamental to the dynamic of multi-agent systems. Agents belonging to a same application need to interact and coordinate their activity to carry out their common global goal, whereas agents belonging to different applications, as in an open scenario, also need to interact, for instance to compete for a resource.

If these interactions are uncoordinated, there is no chance that they lead to the achievement of the common goal, and the role concept is just the one that relates the interactions performed by agents and the objectives of the system. A role is a specific contribution to the system that realizes a part of the global goal, and it determines how this sub-goal may or must be achieved. Thus, roles are basic buildings blocks for defining the organization of multi-agent systems, together with the behavior of agents and the requirements on their interactions [18]. Modeling interactions by roles allows a separation of concerns by distinguishing the agent-level and system-level concerns with regard to interaction. An important characteristic of real-world agent systems is

that an agent may have to change the role it plays over time. If some flexibility constraints require some variety of these roles, agents have to adapt their architecture and functionality as they adopt new roles. These additional capabilities must be dynamically acquired because only a few roles can be hard-coded into an agent. As a matter of fact, this dynamic acquisition is the only possibility in open system where agents enter and leave at will. While designing the overall organization of a system, it is valuable to reuse roles previously defined for similar applications, especially when the structure of interaction is complex. To this end, roles must be specified in an appropriate way, since the composition of independently developed roles can lead to the emergence of unexpected interaction among the agents.

On the other hand, Component Based Development (CBD) [26] promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. It is one of the most important among the recent technical initiatives in software engineering. In CBD, software systems are built by assembling components already developed and prepared for integration. Therefore, the specification of components is useful to both components users and components developers. The specification provides a definition of the component's interface and it must be precise and complete for users; for developers, the specification of a component also provides an abstract definition of its internal structure. The verification of such a well-established specification is needed for a safe composition of systems from components. Verification and CBD are synergistic: CBD introduces compositional structures, and composition rules to build systems, whereas specification along with verification enable the effective development of reliable component-based software systems.

It appears that the facilities brought by the CBD approach fit well the issues raised by the use of roles in MASs, and this paper makes a proposal in this way. It presents the RICO (Role-based Interactions COmponents) model for specifying complex interactions based on roles in open MAS. Although the concept of role has been exploited in several approaches [1, 2, 3, 4, 5, 7, 8, 18, 28] in the development of agent-based applications, no consensus has been reached about what is a role and how it should be specified and implemented. RICO proposes a specific definition of role, which is not in contrast with the approaches mentioned above, but is quite simple and can be exploited in specifications, validations and implementations. In RICO, a role includes a set of *interface elements* (either attributes or operations, which are the provided and required features necessary to accomplish the role's tasks), a *behavior* (interface elements semantics), and *properties* (proved to be satisfied by the behavior). When an agent intends to take a role, it creates a new component (i.e. an instance of the component type corresponding to this role) and this role-component is linked to its base-agent. Then, the role is enacted by the role-component and it interacts with the role-components of the other agents.

This paper focuses on the integration of specification and verification of roles into the Component Based Development. Section 2 defines requirements for modeling role-based interactions as components together with the RICO (Role-based Interactions COmponents) specification model for complex interactions based on roles. Section 3 presents the formal specification language COO (CoOperative Objects) [23], together with SYROCO [24] (an acronym for SYstème Réparti d'Objets CoOpératifs),

an environment that implements COOs. In section 4 we map the proposed RICO specification model to the COO formalism, and specify properties of role-components. An example of interaction protocol is studied to illustrate our approach. We present related approaches in section 5 before to conclude in section 6.

2 Specifying Role-based interactions as Components

In the following, first we overview the specifications of software components in Component Based Software Engineering, and then we present the RICO model for specifying agent roles in open multi-agent systems, which is a template that can be instantiated on various concrete computation model. The main objective is to use the CBD approach for specifying role-based interactions as reusable components [10], and which can be combined together by matching their respective needs and services.

2.1 Specification of Software Components

Component specification is an important issue in CBD. Although this problem has been addressed from the very beginning of the development of component models, it remains one of the challenging problems of Component Based Software Engineering.

Up to now, the specifications of components used in practical software development are limited to syntactic specifications. These specifications include the specifications used with technologies such OMG's CORBA [20] or Sun's JavaBeans [25]. The first of these uses different dialects of IDL (Interface Definition Language) [9], whereas the second uses the Java programming language to specify component interfaces. Therefore, the specification of a component consists of a precise definition of the component's operations and context dependencies, and an essential feature of most component specification techniques is the independence of interfaces from the component implementations.

An important aspect of interface specifications is related to the principle of substitution of components. A component can be substituted if the new component implements at least the same interfaces as the older component. For substitution of components to be safe, however, several techniques try to provide semantics specifications, and most of them use UML [19] and its Object Constraint Language (OCL) [27]. Thus a component implements a set of interfaces, each of which consists of a set of operations. In addition, a set of preconditions and postconditions is associated with each operation, which often depend on the state maintained by the component [17]. Nevertheless this is in general insufficient for the re-usability and extendibility of components. Instead, the semantics or behavior of a software component has to be included in its specification, and the safe substitution of components needs to compare their behaviors [10].

2.2 RICO Specification Model

RICO (Role-based Interactions COmponents) is a role-based interactions abstract model for the specification of roles as components in agent-based applications. The main motivation for modeling role-based interactions as components is to capture interaction patterns that:

- feature well-defined and proved properties,
- may be composed the ones with the others so that the resulting behavior allows to realize an intended goal,
- may be dynamically linked to agent and dissociated when this is no longer necessary,
- enable a separation of concerns by distinguishing the agent-level and system-level with regard to interaction.

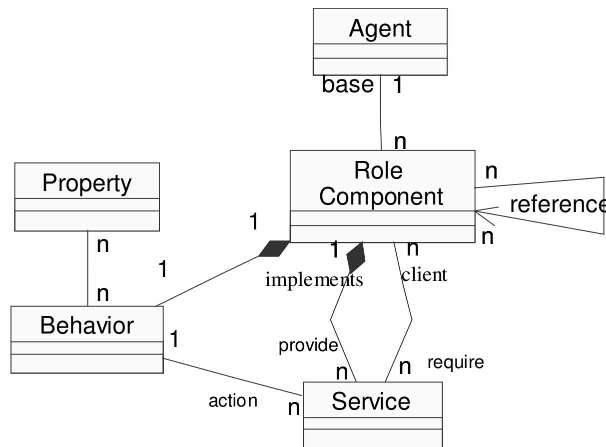


Figure 1. UML metamodel of the concepts used in RICO specification model.

The concept of role has been used in several multiagent development methodologies for modelling and analysing complex system applications. Although there are several role definitions, in all approaches roles are used to identify some task, behaviour, responsibility or operation that should be performed within the system [3]. In RICO, a role is considered as a component providing a set of interface elements (either attributes or operations, which are provided or required features necessary to accomplish the role's tasks), a behavior (interface elements semantics), and properties (proved to be satisfied by the behavior). Figure 1 is a UML class diagram showing the concepts used in RICO and the relationships between them. This UML metamodel specifies that a role-component requires and provides some services, which must be implemented by others role-components, and a service is implemented by one role-component. This independence of services from the role component implementations is an essential feature of RICO specification model according to the CBD approach. Definition 1 gives the explicit definition of the concepts used in RICO. This model is a generic representation of the relationships between these concepts, since in practice,

the expression of these relationships varies from one specification technique to another, that is, one can distinguish between object-oriented specifications and procedural specifications.

In RICO, agents may take up one or several roles simultaneously, and an agent can assume the same role several times, in the same conversation or in distinct ones. For the simplicity, and in order to avoid the conflict access to the agent's resources (for instance agent's public attributes), we assume that only one role is active at each moment in time, and this later is under the agent's decision [3]. Since we are interested in specifications and verification of roles, we assume that the relationship between agent and role is static, that is agents take up roles statically and not dynamically.

Definition 1

A Role Component for a role R is a n-tuple RC = (Ag, Ref, Serv, Behav, Prop), where:

1. *Ag* is the identity of the *base* agent to which RC is linked: Ag has created RC, and RC enacts the role R on its behalf and under its control.
2. *Ref* is a list of role component identities, the role components in the system that RC knows and with which it may interact.
3. *Serv* is the interface of RC, the set of public features through which it interacts with the role components registered in Ref. These features are either *attributes* or *operations*, *methods* according to the standard object-oriented denomination. In addition Serv' features are either provided or required. *Provided* features are maintained and operated by RC itself at the disposal of other role components; provided attributes may be read and provided operations may be called. *Required* features are features provided by other role components and used by RC; the proper behavior of RC needs these attributes and operations and thus depends on their proper behavior. Notice that the interface of a role component forms the basis for its interaction with the environment (agent holding that role and other role components)
4. *Behav* defines the behavior of RC with regard to the other role components of the system. It describes the life-cycle of RC and the sequences of observable actions supported by RC as either the caller of a required operation or the callee of a provided operation; defining Behav in this way assumes that there is no constraint on the access to public attributes; if not, the availability of these attributes must also be specified in any manner. Serv together with Behav may be considered as the functional specification of RC, Behav being a language defined on the operations of Serv, and concerns the Serv's elements by capturing their precise behavior. For instance, the semantics may be specified by using pre- and post-condition associations, describing namely the life-cycle of the role component: sequence, synchronization, and concurrency of operations. Notice that the definition of a role component may also be completed with the mention of private attributes and operations and Behav may includes unobservable actions (private operations), and therefore encapsulates the implementation of the component.
5. *Prop* is a set of behavioral properties that are proved to be satisfied by Behav, so that components requiring the services provided by RC can trust in their fulfillment. Safety and liveness properties are of specific interest [15]: safety properties are invariants that states "nothing bad happens". In contrast, liveness properties state

"something good happens". They are a functional specification of RC that is more abstract than Behav, more declarative in that they are just statements of properties that are guaranteed to be fulfilled by RC in any case. Role's functional properties are useful for selecting a role component and for assessing its suitability, usability or reuse, relative to a given application.

The execution semantics of the Role Components is defined as follows: when an agent executes, if the agent has taken up one or several roles, then at any moment exactly one RC executes (interleaving semantics). Role Components interacts with each other through the call of provided operations. Messages input or output by RC are consumed or generated by Behav through the interface Serv of the Role Component. Role Components allow a proper means for modelling agents and complex interactions, since:

- Roles components are *reactive*, *proactive*, and *autonomous*: through their interface, reactivity is dealt with by operations and services provided, proactiveness is dealt with by services required, and finally a role component execute autonomously according to its behavior Behav, which can be non deterministic. Thus role components can be used as agent-building blocks.
- Considering role components as the active members of protocols facilitates the modeling of the behavior of complex interaction protocols, especially open and concurrent ones. Thus, an agent can play one or more roles at the same time in different conversations (protocols occurrences), and each participation is managed by a specific role component.

In this paper we are interested in an object-oriented specification and implementation of the RICO model. We focus on the specification of these role components, and their implementation by a concurrent formal object-oriented language, the Cooperative Object (COO) formalism. With respect to the specification and design, we have a similar view as, for example, the Gaia methodology [28], and the main focus of this paper is the specification, verification and the implementation of roles.

3 The CoOperative Objects Formalism

In this section we present the COO formalism, a fully concurrent object-oriented formal specifications language, and its implementation SYROCO, together with an example of interaction protocols.

3.1 COO Definition

CoOperative Objects (COO) [23] is a formal specifications language allowing to model a system as a collection of active objects cooperating through an asynchronous request / reply protocol. Each object, being an instance of its COO class, has an identity usable as a reference, and may be dynamically created and deleted.

The structure of a COO includes a set of *attributes*, a set of *operations*, a control structure net called its *OBCS* (Object Control Structure), and a set of *services* supported by this OBCS. The definition of a COO class is divided into two parts, (see an example in section 3.2): the *specification* part concerns the public provided items composing the interface, while the *implementation* part includes the private items, notably the OBCS, a Petri net with objects [16] defining its control structure. Services are public provided methods – with typed in- and out-parameters – and service requests are processed according to the state of the OBCS.

The OBCS of a COO is a high-level Petri net made of transitions, places and arcs, each of them labeled with inscriptions referring to the processed data. Places are state variables whose values (multi-sets of tokens referred to as their marking) determine the current state of the object. Transitions correspond to actions that the object is able to perform, and the occurrence of a transition produces a change in the net's marking. Arcs from places to a transition determine the enabling condition of the transition, while arcs from a transition to places determine the result of its occurrence; the variables labeling arcs surrounding a transition are formal parameters defining how a transition occurrence moves tokens from input to output places. In addition, the value of the tokens linked to these variables at an occurrence of the transition, can be accounted for by means of a guard and processed by means of a piece of code, the transition's *action*. The action of a transition – written in any sequential object-oriented programming language – has access to the attributes and operations of the object as well as to the public attributes and services of other objects.

Some arcs feature a particular shape with the following semantics:

- Inhibitor arc, with a circle on the transition side: the transition is enabled only if the place contains no token (transition t2 in figure 2);

- Place-to-transition clearing arc, with a double arrowhead: each occurrence of the transition removes all the tokens from the input place. An integer value labeling the arc indicates the minimum number of tokens necessary to enable the transition (transition t1 in figure 2).

In this paper, we consider services just with a message sending semantics instead of the full asynchronous request–reply semantics. Each provided service is associated with transitions which process the receipt of requests for that service (*accept-transition*), shown by a pending input arc labeled by the service's in-parameters. Thus, a request for a service is processed by occurrences of transitions in the course of the execution of the OBCS, and it is processed in different way according to the accept-transition that takes the request-token. From the client side, a request for a required service is issued by a *request-transition*. It is distinguished by a pending output arc labeled by the service's actual parameter and its occurrence gives a token to the accept-transitions of the requested service.

The activity of a COO instance consists in processing the calls for its provided operations upon request, and in executing its OBCS as a background task: while transitions are enabled under the current marking, it selects one of them and makes it occur.

The global behavior of a COO system results from the concurrent execution of its constitutive objects. Usually, we need to compose their OBCS for the analysis; this composition is in asynchronous way according to the message sending semantics of service rendering: given a client COO and a server COO, the composition of their

OBCS consists in connecting, through communication places, the request-transitions and the accept-transitions for the same service: each provided service goes with an *entry-place* for receiving the requests. Then, a Client is connected with the service provider through this communication place by an arc from each request-transition towards the suitable entry-place and an arc from the suitable entry-place towards each accept-transition of the service.

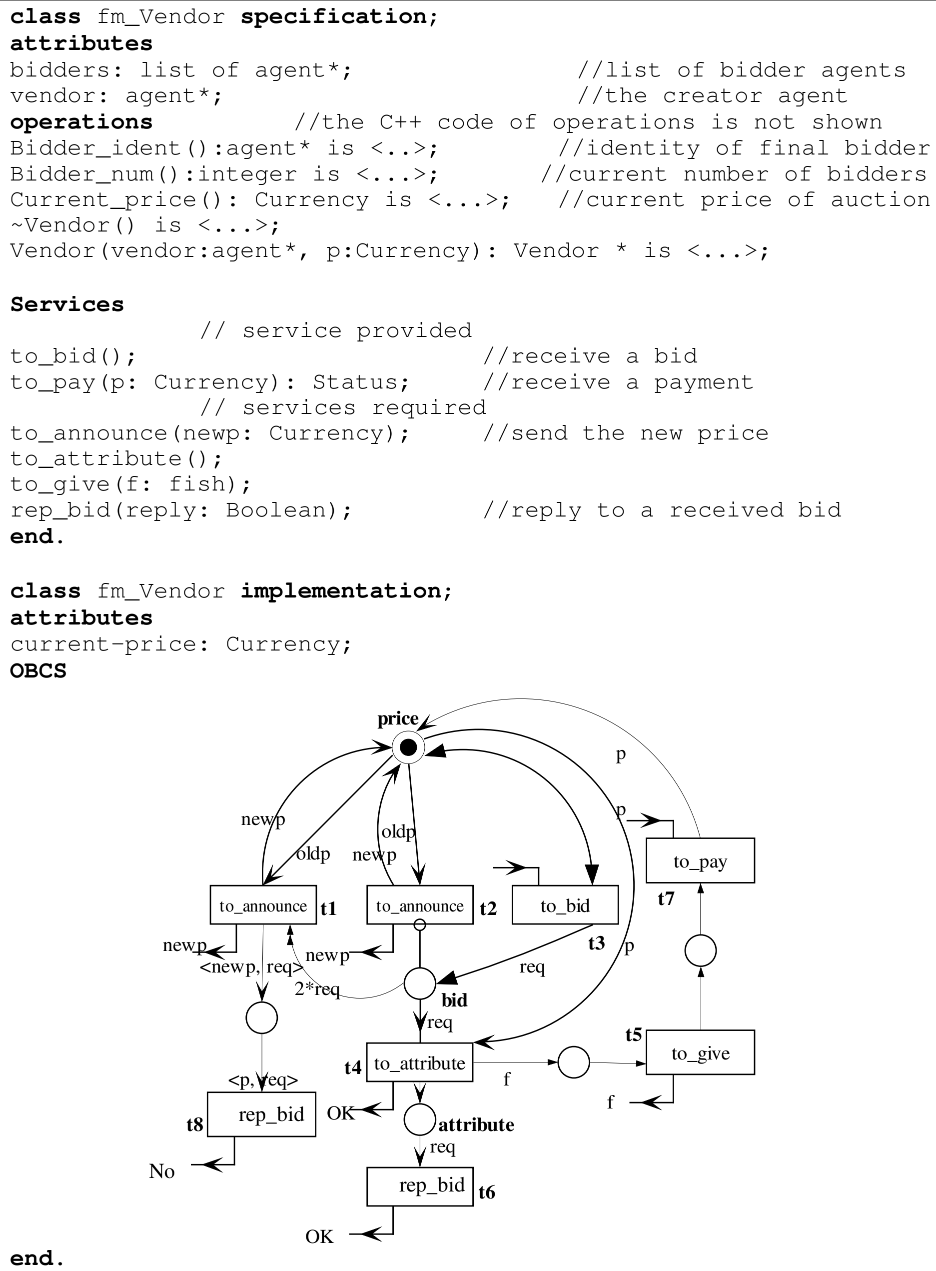
The COO formalism is supported by SYROCO [24], an environment that makes it possible to edit COO class and to generate a C++ class for each COO class, for sequential computing environments (interleaving semantics among objects), for environments supporting threads and also for distributed computing environments compliant with CORBA (true parallelism semantics among objects). SYROCO offers symbolic debugging facilities allowing the designer to examine the state of the OBCS [24], that is the sate of the object (history of transitions occurrence, the previous and the next transition occurrence, value of tokens…). This debugger does not deal with the code of actions, but with the behavior and cooperation among Objects. Each COO has it own debugger, and it is possible to call the debugger of any object from the debugger of another one.

3.2 An Application Example

To illustrate the COO formalism, we will study an example of interaction protocols, the fish-market auction protocol. In [11], authors show how to model this protocol by means of a COO class. In this paper, we show how to design the COO classes that model the two roles, that is, the Vendor and Buyer roles. In any *conversation* following the rules of this protocol, we have a single vendor, and a number of potential buyers, the bidders. The vendor has a bucket of fish to sell for an initial price. A buyer can make a bid to signal its interest. If no (or more than one) buyer is interested, the vendor announces a new lower (or higher) price. When one and only one buyer is interested, the vendor attributes fish to that bidder. Once the bucket of fish is attributed, the vendor gives the fish and receives the payment, while the buyer pays the price and receives the fish.

First, let us consider the fm_Vendor class; figure 2 gives its specification and implementation as a CoOperative Object class. The behavior (Control structure) of this class is as follows: under the initial marking (one token in the `price` place), only the `t2` and `t3` transitions are enabled and may occur. They are respectively the request-transition and accept-transition of the `to_announce` and `to_bid` services, and the vendor may process only these services. The acceptance of a `to_bid` service (by transition `t3`) produces new token into the `bid` place; the transition `t3` remains enabled as long as there is a token in the `price` place, that is until an occurrence of the `t4` transition caused by a request for the `to_attribute` service. This service is accepted if there is exactly one token in the `bid` place, that is there is a single current bidder. The occurrence of `t4` returns `OK` to the buyer, and enables both the request-transition `t5` of the `to_give`, and the request-transition `t6` of the `rep_bid` services. The occurrence of the `t6` transition returns `Ok` to the buyer, and the occurrence of `to_give` service enables the accept-transition `t7` of the `to_pay` service. The

final state of the conversation is reached since the marking of the OBCS becomes the initial one, that is one token in the `price` place.

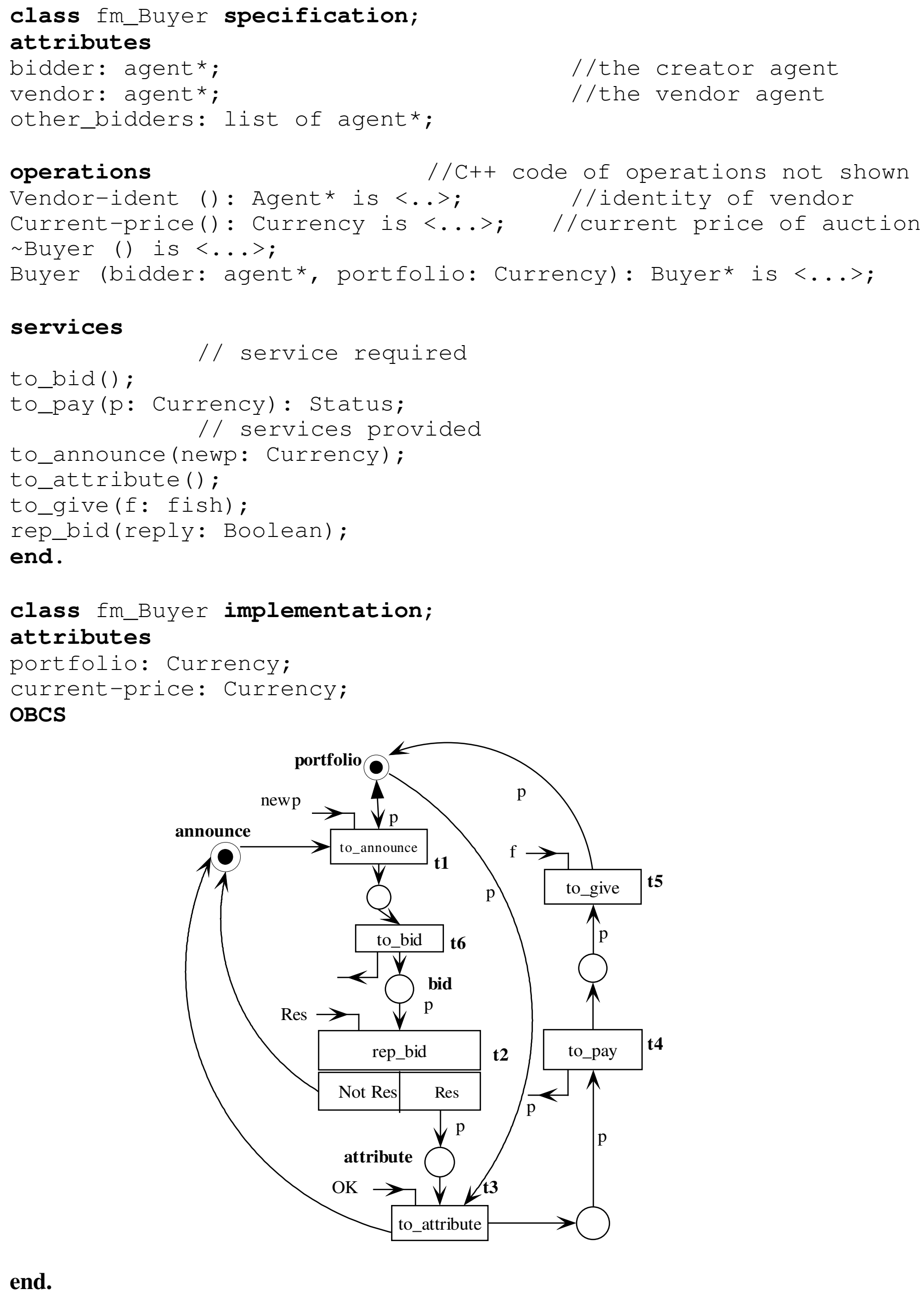


```
class fm_Vendor specification;
attributes
bidders: list of agent*;               //list of bidder agents
vendor: agent*;                        //the creator agent
operations          //the C++ code of operations is not shown
Bidder_ident():agent* is <..>;        //identity of final bidder
Bidder_num():integer is <...>;       //current number of bidders
Current_price(): Currency is <...>;   //current price of auction
~Vendor() is <...>;
Vendor(vendor:agent*, p:Currency): Vendor * is <...>;

Services
             // service provided
to_bid();                        //receive a bid
to_pay(p: Currency): Status;     //receive a payment
             // services required
to_announce(newp: Currency);     //send the new price
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);         //reply to a received bid
end.

class fm_Vendor implementation;
attributes
current-price: Currency;
OBCS
```



```
end.
```

Figure 2. The Vendor role in the fish-market protocol as a COO class, fm_Vendor.



```
class fm_Buyer specification;
attributes
bidder: agent*;                          //the creator agent
vendor: agent*;                          //the vendor agent
other_bidders: list of agent*;

operations                     //C++ code of operations not shown
Vendor-ident (): Agent* is <..>;         //identity of vendor
Current-price(): Currency is <...>;   //current price of auction
~Buyer () is <...>;
Buyer (bidder: agent*, portfolio: Currency): Buyer* is <...>;

services
              // service required
to_bid();
to_pay(p: Currency): Status;
              // services provided
to_announce(newp: Currency);
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);
end.

class fm_Buyer implementation;
attributes
portfolio: Currency;
current-price: Currency;
OBCS
```



```
end.
```

Figure 3. The Buyer role in the fish-market protocol as a COO class, fm_Buyer.

In figure 3, we give the fm_Buyer class. The behavior of this class is as follows: under the initial marking (one token in the `portfolio` place), only the `t1` transition

is enabled and may occur. It is an accept-transition of the `to_announce` service. The acceptance of the `to_announce` service produces an occurrence of the `t6` transition that requests the `to_bid` service. Then, transition `t2` accepts the rep_bid service from the vendor and receives the Res reply. If the value of `Res` is true, it produces a new token into the `attribute` place, and enables the accept-transition of the `to_attribute` service; thus the occurrence of the `to_attribute` service enables `t4`, a request-transition of the `to_pay` service. The rendering of the `to_pay` service enables the accept-transition `t5` of the `to_give` service. Otherwise, if the value of `RES` is false it produces a token into the `announce` place, and the buyer should wait for a new announce from the vendor. The final state of the conversation is reached since the marking of the OBCS becomes the initial one, that is one token in the `portfolio` place, and one token in the `announce` place.

4 Implementation of RICO Model as COO

In this section we show how to map the proposed RICO specification model to the COO formalism, together with some characteristic properties, and give some safety and liveness property of role components and their verification.

4.1 Mapping RICO Model to COO

The mapping of RICO model to COO consists in modeling Role Components as COO classes, both at the *instance* and *type* levels. Referring to definition 1 of a Role Component RC, we have:

- Ag and Ref are registered in attributes, either in the specification or in the implementation according to visibility considerations.
- The provided features are exactly the element declared in the specification part; concerning the required features, services are the ones attached to the request-transitions of the OBCS such as e.g. the services `to_bid` and `to_pay` of transitions `t6` and `t4` in Figure 3, while attributes and operations can be explicitly listed as comment.
- The behavior of RC is defined by its OBCS that rigorously determines (1) when a reception of a request for a provided service can be taken into account and processed by an accept-transition occurrence and (2) when a request for a required service is issued by a request-transition occurrence. The capabilities and the needs wrt to message receptions and sending are thus formally expressed by the RC's OBCS. Other interactions among role components are never constraining and thus have no effect on their respective behaviors, that is: public attributes are continuously readable and calls for operations are synchronously processed upon request.
- The behavioral properties are properties of the OBCS and their technical statement may follow a variety of expressions, some are given below.

As mentioned by Kristensen [14], the concept of roles in object-oriented modeling, should support a set of characteristic properties; the specification and the implementation of RICO model as COO support these properties as follows:

- Visibility/ Dependency: visibility is supported by distinguishing a specification part concerning public items of the interface (operations and services), and implementation part concerning private items, notably the OBCS. Dependency is supported by the fact that the existence of a Role Component depends on that of the agent playing this role.
- Identity/ Dynamicity: supported by the fact that each role component as COO, being an instance of its COO class, has its own identity that can be used as reference, and may be dynamically created and deleted by the agent playing this role.
- Multiplicity/ Abstractivity: supported by the fact that the role components are mapped to COO classes, distinguishing between Role Components on *instance* and *type* level. Thus, several instances of role components may exist for a role at the same time.

Clearly the COO language is not the only way to implement the RICO model, RICO can be supported by any language allowing to explicitly :

- identify and characterize elementary interactions (for Serv) ;
- describe formally a control structure for Behav ;
- have operational semantics in order to deduce more easily an executable implementation from the specifications.
- have compositional semantics in order to deduce emergent interaction among Role Components.

For instance, the type of automatons shown in figure 4 and 5 (section 4.2), could be used to describe the behavior of fm_vendor and fm_buyer classes. The main advantages to use petri nets is that they are completely compositional, that is the composition of the OBCSs (for different roles components) is also an OBCS, even if role components enter and leave the conversations at will [23]; while the definition and the semantics of communicating automatons are more difficult than those of simple automatons, and structural modification of the system (e.g. create, insert and delete an automaton) are more difficult to be taken into account. This mechanism of composition is essential to verify properties related to emergent behaviour when independently developed components roles are put in interaction: the properties of the system are deduced from the properties of its components.

4.2 Safety and Liveness Properties of Role Components

Since the life-cycle of role components is modeled by means of Petri nets, two kinds of properties should be verified [16]: structural property which are related to Petri nets topology. These kinds of properties help designer to build correct specification of the role, independent of the number of agents, and resources; and behavioral properties, that depend on the fixed initial state, and concern qualitative behavior. In this paper, we are interested in safety and liveness behavioral properties of roles and their verification.

Usually, it is possible to generate the reachability graph of the Petri net [16], using tools such as INA [22]. The reachability graph shows all the reachable states and all the sequences of transitions occurrences that may be performed. If the reachability graph is infinite, due to the infinity of a domain value, or to the fact that an action can be repeated again and again (for instance, transition `t3` of `to_bid` service in the Vendor component, figure 2), the covering graph is generated instead; it is finite, and allows the analysis of some safety and liveness properties of the net too.

Safety properties. *Safety* properties of roles are requirements on finite execution. That is, statements of the form “nothing bad happens”. For instance, a specific attribute in a role component is always initialized before this role is taken by an agent. In the Fish-market protocol example, the identity and the initial price of the fish must be fixed by the agent taking the Vendor role (agent who initiates the protocol), that is attributes {vendor, current-price} are not null. These properties can be specified by means of predicates, expressed over the variables listed in the interface of RC. Safety properties express requirements which refer not only to several such status fields at once, but also to a history of states. For instance, using the covering graph of the fm_Vendor’s OBCS shown in Figure 4 (the symbols ! and ? are used to indicate respectively the required and provide services), we technically verify requirements that can be worded in the following way:

- “`to_announce` service may be performed from the initial state or, if, since its previous occurrence, no or more than one `to_bid` service has been performed”.
- “`to_attribute` service may be performed exactly once when, since the previous `to_announce` occurrence, a single `to_bid` intervention has been performed.
- “`to_give`” and “`to_pay`” may be performed only once.

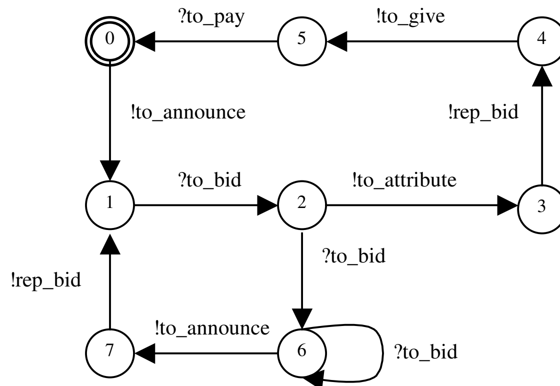


Figure 4. The covering graph of the Vendor in the fish market protocol.

In addition to this classical behavioral analysis, the flexibility and the openness of SYROCO allow to add to the definition of a COO class new attributes to extend the structure of the tokens of any place of the OBCS, and to integrate new methods (functions) to be executed at each occurrence of transitions, namely when the interpreter of

the OBCS removes tokens (from) or adds (into) the places. Besides, it is always possible to add, in the modeling of a COO class, transitions or places whose roles are not functional but only for supervising and detecting particular situations, and namely to be used to check safety properties of the Role Component .

Liveness properties. As mentioned above, safety properties are a very powerful way to guarantee the correctness of a role by verifying that it never reaches an erroneous state. Sometimes this is not enough, and we need to claim that "something good eventually happens". This is the aim of the *liveness* properties. These subtle properties require checking for specific cycles in the reachability graph. So, a liveness property is violated if there is an infinite execution (trace) where progress is not guaranteed; usually, this means that some actions can be repeated infinitely, and the same states of the Role Component are visited again and again. In our example, the analysis of the fm_Vendor's OBCS, tells us that the initial state of the Vendor can be reproduced, and since the initial marking represents the state where there is no ongoing (active) conversation, this reversible property proves that every conversation can be eventually completed and finished. Besides, by exploring the reachability graph of the fm_Buyer's OBCS shown in Figure 5, we can verify requirements such as:

- "after a `to_attribute` intervention, `to_pay` intervention may be performed".
- "after a `to_pay` intervention, `to_give` intervention may be performed".

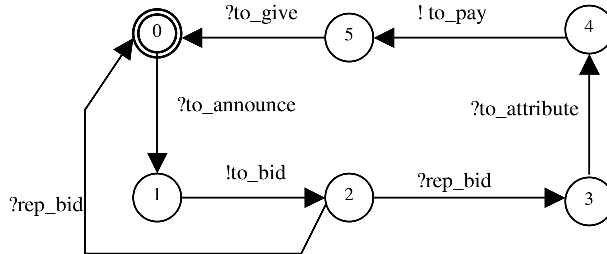


Figure 5. The reachability graph of the Buyer in the fish market protocol.

To further prove additional behavioral properties of roles, INA tool also provides some *state-based* model checking capabilities, and allows us to state properties in the form of CTL formulae [6]. These formulas are defined on the marking of places; so we can specify and verify some key temporal properties about roles on the whole reachability graph of the *Vendor--Buyer* OBCS, the Petri net with Objects obtained by the composition of the OBCS of the fm_Buyer and fm_Vendor classes according to the message sending semantics. For instance:

- *Mutual exclusion*, which is a safety property: for instance in the Vendor-component, no more than one buyer is selected to attribute the fish; it is expressed by the fact that it is impossible to mark both places `bid` and `attribute` at the same time. This property is expressed by:

  ```
  EF(Vendor.bid & Vendor.attribute)
  ```

  that results in `FALSE` if it is impossible to mark both places `announce` and `attribute` at the same time.

- *Concurrency* between role components, which is a liveness property: for instance there exists a path in the reachability graph of the *Vendor/Buyer_OBCS*, in which the `price` place (in fm_Vendor's OBCS), and the `bid` place (in fm_Buyer's OBCS) are marked at the same time, and then, `to_announce` and `to_bid` services, may be executed concurrently. This property is expressed by the holding of the following formula: `EF(Vendor.price & Buyer.bid).`

5 Related Work

There are many approaches and methodologies for the specification of roles (interactions) in multi-agents system. In recent years, roles formation, configuration among roles, and static semantics of roles have been proposed [2, 7]. [7] proposes a meta-model to define models of organizations, based on three concepts: agent, group, and role; agents belong to groups where they hold roles, and interactions take place only between agents member of the same group and according to their respective roles. Our approach is in the same line, since it is based on roles, and the agents that hold a role in the same conversation of a protocol constitute a group. However, our approach gives a formal and precise definition of the interaction patterns – protocols and roles – and groups are defined on the basis of conversations, i.e. occurrences of protocols. In [2], authors study the conditions under which an agent can enact a role and what it means for an agent to enact a role. They define possible relations between roles and agents, and discuss functional changes that an agent must undergo when it enters an open agent system. This work completes our approach, and one can use the proposed relations as constraints interaction requirements that the agents that take up the role must meet. In [3], they argue for the importance of enactment/deactement of roles by agents in multiagent programming, in particular when dealing with open systems. This work study the dynamics of roles in terms of operations performed by agents.; their formalization is conceptually based on the notion of cognitive agents, and therefore, we claim that it can be easily exploited in our specification and implementation of role components. In [28], Gaia methodology adopts an abstract, semiformal description to express the capabilities and expected behaviors of roles involved in protocols. This work is close to ours, since it is based on the organizational abstractions for analysis and design of complex and open interactions, but one possible limitation, is the formal specification, validation and namely the implementation of roles. This is due to the fact that, the life-cycle of roles in Gaia is only expressed by safety and liveness properties, and this methodology does not directly deals with formal analysis and implementation issues.

The Aspect Oriented Programming (AOP) approach has been exploited to implement the concept of roles in [12]. The author discusses the relevance of modeling roles for agents systems. In our approach, roles are considered as components for the interactions between agent's applications. Then, a Role can be selected and reused, considering not only its functionality but also its behavioral properties. In [5], the authors go beyond these AOP's considerations, and propose an interaction model based on the notion of XRole (XML Role), where notations based on XML are

adopted to support the definition and the exploitation of roles at different phases of the application development. This work is close to ours, since it is based on the separation of concerns introduced by AOP, but it suffers from the lack of a formal semantics and as a consequence the possibility to make verification and validation. Thanks to their XML-based syntactic definition, XRole focuses on flexibility, openness, and adaptability of the roles, but not on their formal specification and verification.

Considering the specification and validation of complex interactions and open software systems, [13] proposes an extension of AUML for the modular design of interaction protocols by composing micro-protocols; the main contribution of this approach is to reduce the gap between informal specification of interaction and semi-formal one by using protocol diagrams (AUML sequence diagrams), a graphical language for designing protocols. Nevertheless, specification and verification of open interaction protocols remains non trivial process. In [4], authors develop a role concept for a modeling approach based on the UML and graph transformation systems. They also provide a run-time semantics for roles on concepts from the theory of graph transformation. This approach allows a convenient model for the concurrency, reactivity, and the autonomy of agents. Nevertheless, engineering issues raised related to the use of roles such as the validation and the verification of agent's behavior. In [1], the specification of the MAS is based on a hierarchy of components, defined in term of input/output and temporal constraints. The proposed framework is developed for specification and simulation of MAS. However, the approach has two drawbacks. First, with this approach it seems difficult to refine specifications to implementation language. Second, the verification technique is limited to model checking. [21] proposes a formal framework using the Z language, where initial units (schemas) of specifications are refined in order to obtain a MAS specification. Nevertheless, this framework does not allow to specify temporal and reactive properties of MAS. In our model these aspects are specified by the behaviour and the properties of the Role Components. [8] proposes a multi-formalism based specification approach, including statecharts in Object-Z classes and proposes a formal framework approach for the prototyping and simulation of MAS. Although this approach allows specification and simulation, it has some limitations. Indeed, the Object-Z part of the specification is not yet executable, and only the analysis by simulation of the statechart specifications is possible.

6 Conclusion and Future Work

The aim of this paper is to show how to exploit the concept of role in engineering agent complex interactions (specification, verification, and implementation). Modeling interactions by roles gives several advantages, the most important of which is the separation of concerns by distinguishing the agent-level and system-level with regard to interaction.

Component Based Development (CBD) promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into

complex applications. Software systems are built by assembling components already developed and prepared for integration.

Starting from the above considerations, we have identified requirements for modeling roles-based interactions as components, and propose RICO (Role-based Interactions COmponents), a role-based interactions specification model whose aims is to specify roles in agent-based applications according to component based approach. Then, we have shown how RICO model can been specified and implemented by a concurrent formal object-oriented language, the CoOperative Object (COO) formalism, that enables the formal specification, analysis and validation of open and concurrent interactions. Finally, we have shown how to specify and check properties of a role component exploiting Petri nets theory: reachability graphs for behavioral analysis, and namely others tools such as INA tool for checking temporal properties.

The next step for this work is to exploit some directions. First, we want to adapt SYROCO environment in order to develop an infrastructure supporting RICO model for real size applications, and namely for open and concurrent interaction protocols. Our intention is to explore the coordination model based on Moderators of conversation presented in [11]. This model fits the organization-centered view of MAS as it strictly distinguishes the agent-level and the organization-level concerns with regard to interaction, and the main advantage of this approach is that it is quite simple, both to use and to implement. Second, we are exploring the formal specifications of the relationships between role components and organizational rules [28] such as compatibility and consistency between agents and roles, and interdependence of roles. Finally, we are going to consider non-functional properties in the specification of role components, such as performance, reliability, security, and environmental assumptions; the specification of non-functional properties is still an open area of research in Component Based Software Engineering, and we believe that it will have an impact on the future of software role components specification.

Acknowledgments. This work is funded by the STIC-CNRS Department, under the grant SUB/2003/076/DR16, in the context of C2ECL (Coordination et Contrôle de l'Execution de Composants Logiciels) action.

References

[1] F. M. T. Brazier, B. Dunin Keplicz, N. Jennings, J. Treur, "Desire: Modelling Multi-agent Systems in a Compositional Formal Framework", International Journal of Cooperative Information Systems, 6:67-94, 1997.

[2] M. Dastani, V. Dignum, F. Dignum, "Role Assignment in Open Agent Societies", AAMAS'03, ACM 2003.

[3] M. Dastani, M. B. van Riemsdijk, J.Huslstijn, F. Dignum, J-J. Meyer, "Enacting and Deacting Roles in Agent Programming", AOSE'04.

[4] R. Depke, R.Heckel, J.M.Kuster, "Roles in Agent-Oriented Modeling", International Journal of Software engineering and Knowledge engineering, vol 11, No. 3 (2001) 281-302.

[5] G. Cabri, L. Leonardi, F. Zambonelli "BRAIN: a Framework for Flexible Role-based Interactions in Multi-agent Systems", Proceedings of CoopIS 2003, 2003.

[6] E. M. Clarke, E.A. Emerson, A. P. Sistla, "Automatic Verification of finite-State Concurrent Systems using Temporal Logic Specifications", ACM Transactions on Programming Languages and Systems, Vol 8, N° 2, 1986, pp244-263.
[7] J. Ferber, O. Gutknecht, "Aalaadin: A Meta-model for the Analysis and Design of Organizations in Multiagent system", ICMAS'98, 1998.
[8] P. Gruer, V. Hilaire, A. Koukam, "Formal Specification and Verification of Multi-agent Systems", ICMAS'2000, IEEE, 2000.
[9] M. Gudgin, "Essential IDL: Interface Design for COM", Reading, MA, Addison-Wesley, 2001
[10] N. Hameurlain. "Formal Semantics for Behavioural Substitutability of Agent Components: Application to Interaction Protocols", From Theory to Practice in Multi-agent Systems, LNAI 2296, Springer-Verlag, pp 131-140, 2002.
[11] C. Hanachi, C. Sibertin-Blanc, "Protocol Moderators as Active Middle-Agents in Multi-Agent Systems", AAMAS, 8, 3, p. 131-164, Kluwer Academic Publishers, 2004.
[12] E. A. Kendall, "Role Modelling for Agent Systems Analysis, Design and Implementation", IEEE Concurrency, 8(2): 34-41, April-June 2000.
[13] J-L. Koning, M-P. Huget, J. Wei, X. Wang. Extended Modeling Languages for Interaction Protocol Design. AOSE'2001, Springer-Verlag, pp 93-100, 2001
[14] B.B. Kristensen, "Object Oriented Modeling with Roles", in Proc. 2nd International Conference on Object-Oriented Information Systems (OOIS'95), pp 57-71, Springer .
[15] Z. Manna, A. Pnueli, "Temporal Verification of Reactive Systems-Safety", Springer-Verlag, 1995.
[16] T. Murata, "Petri Nets: Properties, Analysis and Applications", In *Proceedings of the IEEE*, Vol.77, No.4 pp.541-580, April, 1989.
[17] B. Meyer, "Object-Oriented software Construction", Upper Saddle River, NJ, Prentice Hall, 1997.
[18] J. Odell, H. V. .D . Parunak, S. Brueckner, J. Sauter, "Temporal Aspects of Dynamic Role Assignment", AOSE'03, Springer. 2003.
[19] OMG, "OMG Unified Modeling Language specifications", Report V1.3, OMG, June 1999.
[20] OMG, "The Common Object Request Broker: Architecture and Specifications", Report V2.4, OMG, 2000.
[21] M. Luck, M. d'Inverno, "A formal Framework for Agency and Autonomy", ICMAS'95, AAAI Press/MIT Press, editor.
[22] S. Roch, P. H. Starke, "INA: Integrated Net Analyzer, Version 2.2", Humboldt-Universitat of Berlin, April 1999.
[23] C. Sibertin-Blanc, "CoOperative Objects : Principles, Use and Implementation", In Petri Nets and Object Orientation, G. Agha, F. De Cindio eds, LNCS 2001, Springer-Verlag. 2001.
[24] C. Sibertin-Blanc et Al., "SYROCO : Reference Manual V7", University Toulouse1, Oct 1996, (C) 1995, 97, CNET and University Toulouse 1.
Available at http://www.daimi.aau.dk/PetriNet/tools.
[25] Sun Microsystems, "JavaBeans 1.01 Specification",
Available at http://java.sun.com/beans.
[26] C. Szyperski, "Component Software-Beyond Object-Oriented Programming", Addison-Wesley, 2002.
[27] J. Warmer, A. Kleppe, "The Object Constraint Language", Reading, MA: Addison Wesley, 1999.
[28] F. Zambonelli, N. Jennings, M. Wooldridge, "Developing Multiagent Systems: The Gaia Methodology", ACM Transactions on Software Engineering and Methodology, Vol 12, N° 3, July 2003, pp317-370.