

An Approach to Configure Interactions from Generic Protocols

José Ghislain Quenum, Samir Aknine, and Aurélien Slodzian

Laboratoire d'Informatique de Paris6,
8 rue du Capitaine Scott,
75015 Paris, France
{jose.quenum,samir.aknine,aurelien.slodzian}@lip6.fr
<http://www.lip6.fr/OASIS/>

Abstract. Agents interactions which are inspired from generic protocols require a special approach for modelling, designing and execution. They raise up issues related to their reusability, configuration and instantiation. Agent-oriented design methodologies do not separate the protocols from the private part of agents architecture. Thus, none of these parts (public and private) are reusable. Furthermore, these methodologies do not provide any guidelines to configure interactions from generic protocols. Hence inconsistencies are hard to anticipate and solve. In this paper, we propose a method to address these issues by automatically deriving agents coordination mechanisms from generic protocols. This method decouples the protocols from the remainder of agents architecture. It also reduces inconsistencies during interactions.

Key words: MAS, Protocols, Interaction, Modelling, Methodology

1 Introduction

Agents in a multi-agent system (MAS) are expected to bring together their competences in order to execute complex collaborative tasks that none of them can achieve individually. To do so, they execute interactions which are most of the time inspired from generic protocols. Agent-oriented methodologies [12, 1, 2, 7] do not provide any guidelines to configure agents interactions from generic protocols. Consider agents, designed by different¹ designers, executing an interaction based on the Contract Net protocol (CNP) [10]. If the exchanged messages are not uniformly interpreted by the agents the whole interaction becomes inconsistent and the collaborative task being executed fails. Furthermore, these methodologies do not require designers to separate the protocols (the public part of agents architecture) from their agents internal model (the private part of these agents architecture). None of these parts are then easy to reuse and any change in one of them might imply to modify the other one. This burden severely limits the development of multi-protocol agents as well as the openness of the MAS.

To address these issues, we developed an approach to automatically derive agents interactions from generic protocols. In this approach we introduced two new models

¹ As MASs are expected to be open systems.

of an agent, which we define as follows: (a) an interaction model which contains the roles an agent can enact during interaction at runtime; (b) a functional model which contains the functionalities the agent commits to achieve with respect to interactions. Both models are of course related since the behaviour exhibited by the roles of the interaction model is achieved by the functionalities which we call methods in our model.

Our approach recommends to start with the design of the functional model and then to look for similarities between its elements and the actions required to produce or handle the messages of the generic protocols the designer wishes to support. Therefore we devised a unification algorithm to find out these similarities by comparing the descriptions of methods to that of actions. In case of failure, an adaptation algorithm translates the unmatched actions specifications to that of methods. When the unification is finally successful, each specialised role is converted into a finite state automaton and stored in the interaction model.

Our approach decouples public and private parts of agents architecture enabling both to be reusable. It dramatically reduces inconsistencies during interactions and finally eases protocols implementations in the agents models.

Section 2 discusses some related work. Sections 3 and 4 describe our approach in detail: the models we proposed, the algorithms and their implementation. Section 5 concludes the paper.

2 Related Work

The method we develop is related to two fields of MAS design: agent-oriented design methodologies as far as interaction design is concerned, and, protocols engineering with respect to generic protocols specifications.

Several agent-oriented design methodologies have been proposed: Gaia [12], Multi-agent Systems Engineering (MaSE) [1], Tropos [3], Prometheus [7], ROADMAP [11], a combination of UML and graphs transformation [2], to quote a few. These methodologies design individual agents as well as the whole MAS. They produce several output models among which agents models and interaction models. But these methodologies do not decouple agents interaction model from the private part of their architecture. They do not focus anymore on generic protocols specialisation in agents interaction models.

To date, several protocols specification formalisms have been proposed from the protocol engineering perspective. [6] and [4] have extended UML to represent agents interaction protocols. But both extensions still have some drawbacks mainly incompleteness in their specifications. Hence, [5] proposed an approach to automatically derive the formal specifications of a protocol building on semi-formal ones given in [6]. [8] also proposed a protocols specification language which extends propositional dynamic logic. This language mainly applies to negotiation in agent mediated electronic commerce. None of these formalisms focus on generic protocols specifications. However, generic protocols require to be represented as patterns so that several specialisations can be possible. Furthermore, the actions required for the protocols are not explicitly specified in these formalisms. To address the lack of generic protocols specifications

and specialisation approach, we developed an approach to automatically derive generic protocols in agents interaction models.

3 Abstract Models

A designer configures an agent's interaction model by looking for similarities between the actions of the generic protocols he (she) wishes his (her) agent to support and this agent's functionalities. Therefore we model the generic protocols and the functionalities. Finally, as an abstract view of the interaction model, we merge both models into a specialised protocol model.

3.1 Generic protocols model

To identify the fundamental elements needed to describe generic protocols, we studied the sequence diagrams proposed in AUML [6]. We identified some concepts which we define first and then give an in-depth description of our model.

Definition 1. *A protocol is a sequence of messages exchange. It is composed of three elements: a set of roles \mathbf{R} , a set of messages patterns \mathbf{M} and a partially sorted sequence of message exchange Ω .*

Definition 2. *Each actor involved in the interaction is a role. A role imposes some constraints on the agents which are to play it. A protocol holds two types of roles: (1) an initiator which launches the protocol and (2) a participant which involves in the protocol later on.*

Definition 3. *An action is an operation a role executes during an interaction. Its execution copes with events which occur in the role's environment. An action can have input arguments and output results. As well these actions generate some other events while completing.*

Definition 4. *An event is a situation which occurs in a role during interactions; for example a message reception.*

Definition 5. *A phase is a consistent part of an interaction inside a role. While the phase goes on some actions are taken. For example a call for proposals production followed by its emission constitute a phase in the CNP.*

An in-depth examination of these concepts help identify the way to combine them. Sequence diagrams fall short in raising only the communication level. Indeed, sequence diagrams hide some mandatory information in the protocols (see [8]). For example, in the CNP the initiator collects all the participants bids and makes a decision once a defined delay has elapsed. This action (called `Deliberation`) actually involves several participants; therefore it does not appear in the sequence diagram of the protocol. We call such actions *global actions*. Moreover, all the data manipulated inside protocols are not encapsulated in a message. For example the deadline indicating when the `Deliberation` action should be taken is not encapsulated in a message. We fixed

this void by enriching sequence diagrams elements with *variables* (to represent data which are not encapsulated in a message) and *global actions*.

The abstract model we propose here is formalised as a set of EBNF rules. But, to enable an automatic derivation, we represented these protocols in XML, thus we derived a DTD from the EBNF grammar. We'll illustrate the model using its xml representation.

1. In this model, a protocol has some *attributes* (class of the protocol, number of participant and return value) and *keywords*. It is composed of roles and message patterns.

```
< protocol > ::= < protocoldescriptors > < roles > < messagepatterns >
< roles > ::= < role > < role > | < roles > < role >
< messagepatterns > ::= < messagepattern+ >
```

2. A role has some *attributes* and *keywords*. It may contain variables and global actions and is decomposed into a set of phases.

```
< role > ::= < roledescriptors > < variables? > < actions? > < phases >
```

The initiator of the CNP has three variables: a collection where it stores the participants bids, a deadline for bids proposal and the deliberations upon these bids.

```
< variables >
  < variable id="bidlist" type="collection" />
  < variable id="deadline" type="date" />
  < variable id="deliberations" type="mapping" />
</ variables >
```

The global actions are represented in the same way as actions. We further give an example of actions description.

3. Actions are classified into several types: appending, updating, removing, setting and computing data. These actions may have input arguments and output results which are described using abstract types (see table 1). There are also actions that send a message and which, obviously, take this message as an argument. Actions are also bound to input and output events. Events, arguments and results can be combined in sets by means of three possible connectors: "and", "or" and "xor". The "and" connector expresses the joint occurrence of the elements while the "xor" expresses alternative occurrence of these elements. Finally, the "or" connector can be perceived as a special "and" where the elements are optional but at least one of them should occur. Note that the combined elements can also be combinations of other single or combined elements. The rules describing an action are shown as follows:

```
< action > ::= < category > < description? > < signature > < events >
< category > ::= "append" | "custom" | "remove" | "send" | "set" | "update"
< signature > ::= < arguments > | < messages >
< arguments > ::= (< argset > | < argdesc >)+
< argset > ::= < settype > (< argset > | < argdesc >)+
< argdesc > ::= < identifier > < type > < direction >
< direction > ::= "in" | "out" | "inout"
< messages > ::= (< message > | < messageset >)+
< messageset > ::= < settype > (< messageset > | < message >)+
< settype > ::= "and" | "or" | "xor"
< events > ::= (< event > | < eventref > | < eventset >)+
```

`< eventset > ::= < settype > (< event > | < eventref > | < eventset >) +`
`< event > ::= < identifier? > < eventtype > < object > < direction >`

As an example, follows the description of the `prepareCFP` action which the initiator uses to produce the *call for proposals*.

```
<action category="custom" description="prepareCFP">
  <signature>
    <arg type="date" dir="in" id="arg3"/>
    <arg type="any" dir="out" id="arg4"/>
  </signature>
  <events>
    <event type="varctn" dir="in" obj="deadline" prv="arg3"/>
    <event type="msgctn" dir="out" obj="cfp" id="evt2" prv="arg4"/>
  </events>
</action>
```

The `provides` (`prv` in short) attribute is used to relate an event to an action's argument or result.

4. A *message pattern* is an abstract description of a message. It holds a communicative act expressed in an agent communication language, a content type and probably a content pattern. Senders and receivers are detected from the flow of exchange.

`< messagepattern > ::= < identifier > < performative > < type > < contentpattern >`

Abstract types	Description
<i>Number:</i>	corresponds to any numerical value.
<i>String:</i>	describes strings.
<i>Char:</i>	describes any ASCII character.
<i>Boolean:</i>	describes a boolean; a variable having two possible values: <i>true</i> and <i>false</i> .
<i>Date:</i>	represents time.
<i>Collection:</i>	describes a collection of data.
<i>Mapping:</i>	associative array mapping a key to a value.
<i>Any:</i>	Most generic data type in our model. Represents any kind of data.
<i>Null:</i>	describes a void data.

Table 1: Abstract types

3.2 Functional model

The functionalities of an agent are represented as methods². Each method is described by its signature, its category (the same as actions in generic protocols) and the constraints placed upon it.

² This is not a restriction to object-oriented method.

Here again, we use abstract data types to describe the signature of a method. But the designer can introduce user data types which should inherit from the built-in types defined in table 1. Unlike in actions signature, there is no combination of arguments or results in methods. Indeed, a method's signature should be perceived as a path in a graph corresponding to the similar action's signature.

We define two constraints on methods. First succession constraints introduce causality between methods. Second exclusion constraints help to prevent a method to be executed once another one has completed. Such methods are of course concurrent.

The functional model is formalised by a set of EBNF rules from which we derived a DTD.

1. In this model, the built-in types as well as the user defined types in use are defined.

```

<types> ::= <bits><udts>
<bits> ::= <bit+>
<bit> ::= "number" | "string" | "char" | ...
<udts> ::= <udt+>
<udt> ::= <udtname><super>(<coludt>|<mapudt>|<strutudt>)?

```

While describing user-defined types, we pay a special attention in the case they inherit from a collection, a mapping or a structure. Indeed, when a user-defined type (udt) inherits from a collection one should indicate its items type. As well, when a udt inherits from a mapping one should indicate the key and value types. Finally when a udt inherits from a structure one should indicate each field's type. Moreover, each udt should explicitly indicate its super type.

Consider the following example:

```

<types>
  <bitypes>
    <bitype id="any" name="any"/>
    <bitype id="collection" name="collection"/>
  </bitypes>
  <udtypes>
    <udtype id="cfp" name="CallForProposal" super="any"/>
    <udtype id="proposal" name="Proposal" super="any"/>
    <udtype id="proposals" name="Proposals" super="collection">
      <collectionudtype itemtypeid="proposal"/>
    </udtype>
  </udtypes>
</types>

```

In this example, we use two built-in types: any and collection. Then we introduce CallForProposal and Proposal which inherit from any. Proposals is a another user-defined type which inherits from collection and contains items of type Proposal.

2. Methods description follows then. For each method, we provide the category, its signature and the constraints placed upon it.

```

<methods> ::= <method+>
<method> ::= <methodproperties><signature><constraints?>

```

```

<methodproperties> ::= <identifier><word><category>
<category> ::= "append"|"custom"|"remove"|"set"|"update"
<signature> ::= <arg+>
<arg> ::= <typeid><direction>
<direction> ::= "in"|"out"|"inout"
<constraints> ::= <constraint+>
<constraint> ::= <status><constraintdescriptor>
<status> ::= "executed"|"notexecuted"
<constraintdescriptor> ::= (<methodid>|<category>
|<typeid><direction>)+

```

As an example let's provide a representation of the `prepareCFP` action the CNP initiator uses to produce the call for proposals and the `handleProposal` action the same role uses to store the bid a participant sends.

```

<method id="m0" name="Method0" category="custom">
  <signature>
    <arg type="date" dir="in"/>
    <arg type="cfp" dir="out"/>
  </signature>
</method>
<method id="m3" name="Method3" category="append">
  <signature>
    <arg type="proposal" dir="in"/>
    <arg type="proposals" dir="inout"/>
  </signature>
  <constraints>
    <methodref status="executed" id="m0"/>
    <methodref status="notexecuted" id="m1"/>
    <methodref status="notexecuted" id="m2"/>
    <methodref status="notexecuted" id="m5"/>
  </constraints>
</method>

```

In this example, `Method0` and `Method3` respectively correspond to `prepareCFP` and `handleProposal`. We further comment these matchings while illustrating the specialisation algorithms.

4 Protocols specialisation

Specialising a role of a generic protocol for an agent consists in replacing generic actions of this role by some functionalities the agent achieves. For this purpose we look for probable similarities between actions and functionalities by means of a unification algorithm. In case of failure, an adaptation algorithm provides the specifications of the missing methods. Unification and adaptation algorithms are improved versions of those proposed in [9].

Algorithm 1 Actions-Methods Unification

```
Input: Set Roles (roles to unify)
Input: Set  $S_2$  (agent's methods)
Result: Map  $m$  ( $key=action$ ,  $value=method$ )
Result: Set  $S$  (unmatched actions)
 $T_m \leftarrow Tree(S_2)$ ;
for all  $role \in Roles$  do
   $T_a \leftarrow Tree(getActions(role))$ ;
   $explorer.add(root(T_a))$ ;
   $candidates.put(root(T_a), root(T_m))$ ;
  while  $explorer$  is not empty; do
     $curact \leftarrow read(explorer)$ ;
     $result \leftarrow match(curact, candidates.get(curact))$ ;
    if  $result$  is empty then
       $S.add(curact)$ ;
    else if  $result$  is a singleton  $\{mth_k\}$  then
      map  $curact$  and  $mth_k$  to their ancestors;
      for all act descendant of  $curact$  do
         $explorer.add(act)$ ;
         $candidates.put(act, children(mth_k))$ ;
         $m.put(curact, mth_k)$ ;
        resolve conflicts;
      end for
    else
      map  $curact$  and  $mth_k$  to their ancestors;
      for all act descendant of  $curact$  do
         $explorer.add(act)$ ;
         $conflicts.put(curact, result)$ ;
         $candidates.put(act, allChildren(result))$ ;
      end for
    end if
  end while
   $S.addAll(conflicts.keySet())$ ;
end for
```

4.1 Unification algorithm

Algorithm 1 matches actions to methods. In this algorithm we construct a tree T_m using the constraints on methods. As well, for each role to be specialised we construct a tree T_a for its actions. We explore nodes in T_a in breadth-first order to limit the comparisons. A list, `explorer`, contains the nodes to explore and a hash table, `candidates`, maps each action to a set of methods it will be compared to. This comparison is achieved in three steps:

1. category: check whether an action has the same category as a method;
2. signature: check if a method's signature is a valid instantiation of that of an action;

- constraints: check whether a method's succession and exclusions constraints match the events of an action. Notice that a method having no successor can match an action which might have ones but can also terminate the interaction.

This matching process returns the set of methods (M) which compare to an action a_i .

if M is empty a_i is inserted in the unmatched actions set.

if M is a singleton a_i is mapped to the only one element m_k of M ; its children are added to the `explorer` list and are mapped to the children of m_k in the `candidates` map. Then we try to resolve conflicts.

if M contains at least two methods this conflict is reported and once any descendant of a_i will match a method we'll resolve it. All the children of a_i are inserted into `explorer` and mapped to all the children of the elements of M in `candidates`

As an illustration of the unification algorithm, we comment the reasons why `Method0` and `Method3` respectively match `prepareCFP` and `handleProposal`.

- each pair (action, method) is of the same category;
- as `cfp` is a subtype of `any`, `Method0`'s signature is compatible with that of `prepareCFP`. As well, `Method3`'s signature is compatible with that of `handleProposal` since `proposal` and `proposals` are respectively subtypes of `any` and `collection`.
- no constraint is placed upon `Method0` and `prepareCFP` is the initial action of its role; `Method3` can be executed if `Method0` has completed but its execution excludes other methods. In fact, `handleProposal` follows `prepareCFP` and can be executed only if none of `closeInteraction`, `handleDenial` and `handleNUD` have been executed.

The unification we described may fail due to the functional model's incompleteness. Therefore, the specifications of the unmatched actions are translated into methods specifications in order to enrich this functional model.

4.2 Adaptation algorithm

Algorithm 2 Actions Adaptation

Input: Set S (unmatched actions)

Result: Set S' (generated methods)

```

for all action  $a_i \in S$  do
  create_method( $m_i, a_i$ );
  set_category( $m_i, a_i$ );
  create_signature( $m_i, a_i$ );
  create_constraints( $m_i, a_i$ );
  insert  $m_i$  in  $S'$ ;
end for

```

Algorithm 2 achieves actions adaptation to methods. In this algorithm each unmatched action is translated into a method. This translation is concerned with the category, the signature and the constraints. The action's category is simply reported in the method. The signature is generated following the three rules below:

$$I(A_j) = \alpha_1 \text{ and } \alpha_2 \dots \text{ and } \alpha_n \implies I(M_l) = \alpha_1 \text{ and } \alpha_2 \dots \text{ and } \alpha_n \quad (1)$$

$$I(A_j) = \alpha_1 \text{ or } \alpha_2 \dots \text{ or } \alpha_n \implies I(M_l) \in \{\{\alpha_1\} \dots, \{\alpha_n\}, \dots \{\alpha_1, \dots, \alpha_n\}\} \quad (2)$$

$$I(A_j) = \alpha_1 \text{ xor } \alpha_2 \dots \text{ xor } \alpha_n \implies I(M_l) \in \{\{\alpha_1\}, \{\alpha_2\}, \dots \{\alpha_n\}\} \quad (3)$$

The α_i are whether single arguments or a combination of such arguments. $I(A_j)$ and $I(M_l)$ represent respectively the input of an action A_j and a method M_l . The rules we defined are recursive and apply to the α_i if they combine arguments. We follow the same rules to derive methods output.

The actions tree T_a helps identify the succession and exclusion constraints. In this tree an edge made up of the current action and its parent translates to a succession constraint, while the siblings of this action share exclusion constraints with it.

To illustrate the adaptation algorithm, let's translate the `Deliberate` action which an initiator uses to deliberate upon the participants bids) to a method, say `Method4`.

```
<method id="m4" name="Method4" category="custom">
  <signature>
    <arg type="date" dir="in"/>
    <arg type="proposals" dir="in"/>
    <arg type="deliberations" dir="out"/>
  </signature>
  <constraints>
    <methodref status="executed" id="m3"/>
    <methodref status="notexecuted" id="m5"/>
  </constraints>
</method>
```

To deliberate, an initiator waits for the deadline and once passed checks whether bids have been proposed. It then achieves its deliberation, accepting or rejecting bids. During this translation, the category is reported. We exploited the action's tree to deduce signature and constraints. Indeed, `Method4` follows `Method3`. Therefore, the collection type is replaced by *proposals*. In addition, we constrain `Method4` to be executed only when `Method3` has completed.

4.3 Final compilation

As another assistance to designers, we convert specialised roles into finite state automata as their execution form. Our automata contain three types of states: an *initial state*, a *final state* and *standard states*. Transitions from one state to another are fired depending on three types of events: *message event* which notifies that a message has been received, *timeout event* which informs that a duration has elapsed and that an action should be taken, *done event* which notifies that a given action has completed with

the expected result therefore an action should be taken. Moreover, some transitions can be immediately fired once a state is reached; we associate such transitions to *true events*.

A role's actions tree T_a guides its automaton's construction. Therefore actions without predecessor become transitions from the initial state, while actions with no successor become transitions to the final state. Finally actions having at least one predecessor and one successor become transitions from a standard state to another. Notice that when an action has several predecessors, it sounds that the transitions corresponding to these predecessors have the same target state.

5 Conclusion

To avoid un reusable agents interaction models and inconsistencies during interactions at runtime, we developed an approach to automatically derive agents interaction models from generic protocols. This approach decomposes into four steps:

1. abstract representation of generic protocols based on AUML sequence. This model emphasises the actions required to produce and handle the messages exchanged during the protocols.
2. abstract representation of the agent functionalities describing the methods the agent will execute during interactions.
3. specialisation of generic protocols matching the actions of the protocols to methods in the functional model.
4. conversion of unified roles into finite state automata which are inserted into agents interaction models.

Our approach has been implemented in Java (version 1.4) and applied to two non-trivial application projects, namely Safir (www.projet-safir.org) and Princip (www.princip.net), which both use MASs to realise sophisticated web pages filtering. The approach has been proved on several protocols: CNP, FIPA Query, FIPA Request, and other protocols designed in the context of these projects like the Safir *Incremental Problem Solving* protocol, the Princip *Association Formation* protocol, to quote a few. Using these protocols, we configure the agents interaction model at design time and the interactions that take place between these agents at runtime simply execute the generated automata.

The generic protocols specialisation is a step in our research. Once roles of generic protocols have been specialised following our approach, it sounds to focus on the question when and how these roles are selected to execute interactions at runtime. A solution to this issue takes our approach one step further.

Bibliography

- [1] S. Deloach and M. Wood. An overview of the multiagent systems engineering methodology. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of the 1st International Workshop on Agent Oriented Software Engineering*, volume 1957. Springer Verlag, June 2000.
- [2] R. Depke, R. Heckel, and J. M. Kster. Formal agent oriented modeling with uml and graph transformation. *Science of Computer Programming*, 2001.
- [3] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos development methodology: Processes, models and diagrams. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, 2002.
- [4] J-L Koning, M-P Huget, J. Wei, and X. Wang. Extended modeling language for interaction protocol design. In *Proceedings of the Agent Oriented Software Engineering*, pages 93–100, 2001.
- [5] H. Mazouzi, A. El Fallah Seghroughni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 517–526, 2002.
- [6] J. Odell, V. D. Parunak, and B. Bauer. Representing agent interaction protocols in uml. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of the 1st International Workshop on Agent Oriented Software Engineering*, volume 1957. Springer Verlag, June 2000.
- [7] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, July 2002.
- [8] S. Paurobally, J. Cunningham, and N. R. Jennings. Developing agent interaction protocols using graphical and logical methodologies. In *Proceedings of the first International Workshop on Programming Multi-Agent Systems*, pages 45–54, Melbourne, Australia, 2003.
- [9] J. G. Quenum, A. Slodzian, and S. Aknine. Automatic derivation of agent interaction model from generic interaction protocols. In *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering*, 2003.
- [10] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. On Computers*, 29(12):1104–1113, 1980.
- [11] J. Thomas, A. Pearce, and L. Sterling. Assembling agent oriented software engineering methodologies from features. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, 2002.
- [12] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.