# Engineering Agent Conversations with the DIALOG Framework

Fernando Alonso, Rafael Fernández, Sonia Frutos, and Javier Soriano

School of Computer Science, Universidad Politécnica de Madrid,
28660 Boadilla del Monte, Madrid, Spain
{falonso, rfdez, sfrutos, jsoriano}@fi.upm.es

**Abstract.** This paper presents the rationale behind DIALOG: a formal framework for interaction protocol (IP) modeling that considers all the stages of a protocol engineering process, i.e. the design, specification, validation, implementation and management of IPs. DIALOG is organized into three views. The *modeling view* allows visual IP design. The *specification view* automatically outputs, from the design, the syntactic specification of the IPs in a declarative-type language called ACSL. This improves IP publication, localization and communication on the Web, as well as IP machine learning by agents. Finally, the *implementation view* provides a formal *structural operational semantics* (SOS) for the ACSL language. The paper focuses on the developed SOS, and shows how this semantics allows protocol property verification and eases automatic rule-based code generation from an ACSL specification for the purpose of simulating IP code execution at design time, as well as improving and assuring correct IP compliance at run time.

## 1 Introduction

Agent communication languages (ACLs) such as the *ARPA KSI Knowledge Query and Manipulation Language* (KQML) [1] and the *FIPA* Agent Communication Language (ACL) [2] are based on the concept of agents interacting with each other by exchanging typed messages that model the desired *communicative action* (e.g. inform, request, response, etc.), also referred to as *speech act* or *performative*, and a declarative representation of its content.

However, agents do not participate in isolated message exchanges, they enter into *conversations* [3], i.e. coherent message sequences designed to perform specific tasks that require coordination, such as negotiations or agreements. Societies of agents cooperate to collectively perform tasks by entering into conversations. In order to allow agents to enter into these conversations without having prior knowledge of the implementation details of other agents, the concept of *interaction protocols* (also known as *conversation policies*) has emerged [4]. *Interaction protocols* (IPs) are descriptions of standard patterns of interaction between two or more agents that may be as simple as request/response pairs or may represent complex negotiations involving a number of participants. They constrain the possible sequences of messages that can be sent amongst a set of agents to form a conversation of a particular type. A number of IPs have been defined, in particular as part of the FIPA standardisation process [5]. The importance of IPs in the design of an agent society is evident not only from their fitness for structuring behavior, but also as an organizational factor [6].

This approach to agent interaction necessarily depends on the provision of a *framework* to support the modeling of interactions between agents that considers all the stages of a *protocol engineering* process, i.e. the design, specification, validation, implementation and management of IPs considered as resources. Some relevant aspects to be taken into account when building such a framework are (a) the ease of modeling the communicative agent behavior,mainly, the behavior of agents that obey complex interaction patterns, (b) protocol maintainability and ease of reuse at both the design and specification level, (c) reliability, from the viewpoint of design validation and property verification and as regards assuring proper protocol compliance by participant agents, (d) availability and accessibility of both the protocols (i.e. designs and specifications) and ongoing conversations (i.e. protocol instances, protocol state and participant agents), being related to agent interoperability, and (e) scalability of both the designs and specifications (ease of composition) and the ongoing conversations for adaptation to large MAS. This paper presents the rationale behind DIALOG: a formal framework developed by the authors which deals with all these aspects at the IP architectural design, formal specification and implementation levels.

The remainder of the paper is organized as follows. Section 2 presents an overview of the DIALOG framework. We then concisely review in section 3 the fundamentals of the ACSL protocol specification language, which is at the core of the DIALOG *specification view*. Section 4 focuses on the *implementation view* and describes the formal *structural operational semantics* (SOS) that has been developed for the ACSL language. Finally, we conclude the paper in section 5.

## 2 DIALOG Framework Overview

The problem of IP specification is not new to *agent societies* developers, and a wide range of solutions have been proposed (cf. [7]). We find, however, that there is a huge void between the existing proposals based on formal techniques, whose design is extremely complex (e.g. Colored Petri Nets [6,8]), and the graphic notation-based techniques (e.g. AUML [9]), which are devoid of precise semantics and rule out automatic specification exchange in a machine readable language and interpretation for the purpose of specification simulation, validation and execution. DIALOG intends to fill this gap by means of three interrelated views:

- The *modeling view* eases the visual design of IPs by means of an AUML-based graphic notation [9]. The proposed notation ($AUML^+$) extends existing AUML and furnishes this notation with formal semantics. The latter is essential for developing the *specification view*. See [10] for a detailed description of this view, which has been ommitted here for the sake of briefness.
- The *specification view* automatically outputs the syntactic specification of an IP from its visual design in a declarative-type language called ACSL. This

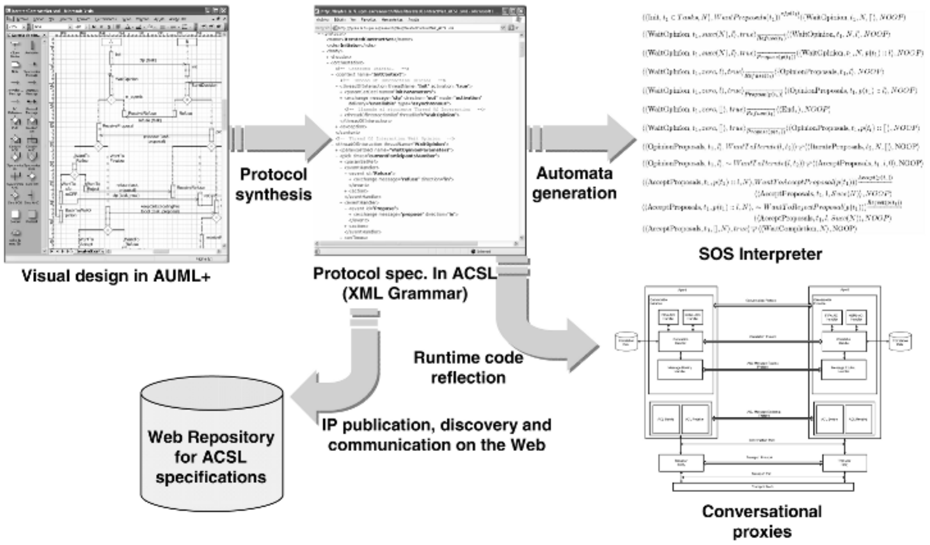
Visual design in AUML+
Protocol synthesis
Protocol spec. In ACSL (XML Grammar)
Automata generation
SOS Interpreter
Runtime code reflection
IP publication, discovery and communication on the Web
Web Repository for ACSL specifications
Conversational proxies

**Fig. 1.** Tools and artifacts of the DIALOG framework

improves IP publication, discovery and communication on the Web, as well as the machine learning of IP by agents. ACSL is an abstract syntax for which an XML grammar has been developed by means of the XML Schema formalism, in order to be able to validate the specifications syntactically, and to make easier their use in Internet environments. A KIF-based grammar is also available, and the mapping between both grammars is trivial by means of an XSLT-based parser. We concisely review the fundamentals of the ACSL language in section 3, see [11] for a more detailed description of this key component of the DIALOG framework.

- The *implementation view* is based on the provision of a formal *structural operational semantics* (SOS) for the ACSL language. The developed formal semantics allows us to verify the properties of the designed IPs, such as their termination in finite time, conversational state reachability or the absence of deadlocks or starvations. On the other hand, the developed SOS automatically outputs rule-based code from the ACSL specification for the purpose of (1) simulating protocol execution at design time and (2) improving and assuring correct IP compliance at run time. Section 4 focuses on this view.

Figure 1 gathers the different products of the IP engineering process and the tools of the proposed framework (consider each product and tool in the figure as a block. The details in each block are not necessary for understanding the figure). These tools allow: (1) the visual composition of IPs in $AUML^+$ notation, (2) automatic ACSL specifications generation (using an XML grammar) for models built in $AUML^+$, (3) the output of a SOS interpreter associated with these specifications, and (4) the generation, by means of code reflection techniques, of conversational proxies that improve IP compliance at run time. Both the $AUML^+$ Editor and the ACSL/SOS Generator are open source tools. The source code is being distributed under GPL license, and is available from [12].

## 3 ACSL Language Fundamentals

The ACSL language defines an abstract syntax that establishes a vocabulary that provides a standard and formal description of the contractual aspects of IPs modeled using $AUML^+$ for use by design, implementation and execution monitoring libraries and tools. ACSL separates internal agent IP implementation from its external description. This is a key point for improving communication interoperability between heterogeneous agent groups and/or agents that run in heterogeneous agencies (platforms). It is based on ACL messages specifying the message flow that represents an IP between two or more agents and requires no special-purpose implementation mechanism.

The overall structure of a protocol specification in ACSL is composed of a *name*, a *header* and a *body*, all defined in the context of a block element *protocol.* The *name* element identifies the protocol for the purpose of referencing from other specifications in which it is to be embedded or with which it is to be interlinked. The *header* element declares the correlation sets and the properties used in the message exchanges for correlation and dynamic linking and to specify the semantic elements, respectively. The *body* of the protocol contains the specification of the basic exchange pattern. This item is formed by the composition of many *threadOfInteraction* elements that fork and regroup to describe the communicative behavior of the agent. The *threadOfInteraction* element is used to directly specify an exchange pattern or reference a protocol definition included in another specification by means of a qualified name (i.e. the conversation is specified in ACSL as IP composition).

A *threadOfInteraction* (Figure 2) combines zero or more atomic actions, references to subprotocols, conditional and iterative constructs and other *threadOfInteraction* that are interpreted sequentially and in the same order in which they are referenced. The sequence finishes when the last element ends.

The following describe such constructs with the level of detail necessary for understanding the remainder of the paper. Use cases of such constructs in ACSL specifications can be found in section 4, as they are needed. See [10] for a more detailed description of the ACSL abstract syntax.

Atomic actions are basic elements upon which an exchange pattern specification is built. ACSL includes four classes of basic actions, as shown in the *actionGroup* element decomposition illustrated in Figure 2: null action (*empty*), message exchange (*exchange*), protocol exception raising (*raise*) and time-outs (*delayFor*, *delayUntil*).

Message exchanges (*exchange* element) are the fundamental atomic actions in agent interaction. ACSL includes only the exchange properties that are part of the protocol specification, according to the ACL approach.

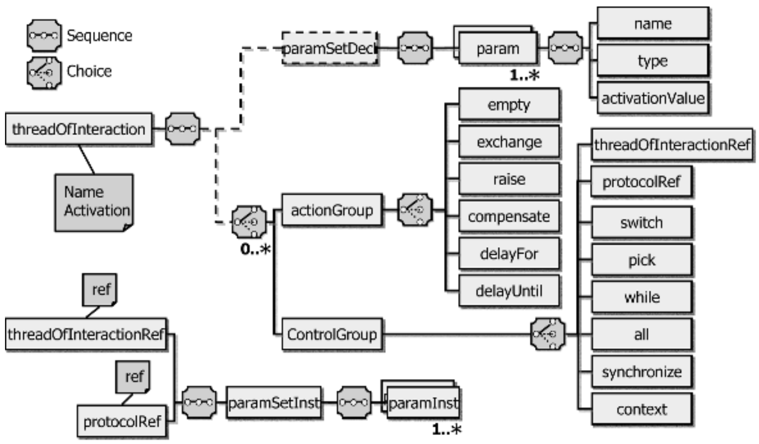

**Fig. 2.** Constructs of the language for specifying an exchange pattern

A *threadOfInteraction* eases the composition of an exchange pattern by means of a set of control constructs (*ControlGroup* in figure 2) that express conditional, concurrent and iterative interaction flows. These constructs are described below.

**Switch:** Expresses a conditional behavior equivalent to the XOR in AUML.
**While:** Repeats the exchange pattern determined by a *threadOfInteraction* an undefined number of times, until the given *condition* is no longer true.
**All:** Expresses the concurrent execution of a set of interaction flows that is not subject to any time order. *All* expresses the semantics of the AND connector in AUML notation.
**Pick:** Expresses precondition waits. It waits for the reception of an event (or set of events) and then executes an exchange pattern associated with this event. The possible events are message reception and end of a *delay* action.
**Repeat:** Repeats the exchange pattern given by a thread of interaction an pre-established number of times. The actual number of times it is repeated is opaque, i.e. is not part of the ACSL specification.
**Synchronize:** Establishes the set of threads of interaction that should be synchronized after an *All*, a multiple-choice *Switch* or an *Or*.

See [11] for a detailed description of the exception- and compensation-handling related ACSL constructs shown in Figure 2.

## 4 Implementation View: ACSL Semantics

The definition of an XML grammar for ACSL by means of the *XML Schema* formalism can only validate the IP specifications syntactically. To be able to validate and evaluate these specifications semantically, the ACSL language also needs to be furnished with formal semantics that can unambiguously describe the dynamic meaning of its syntactic constructs.

The provision of formal semantics for ACSL means that the IP specification can be analyzed to find out whether the IP has certain properties, such as termination in finite time, conversational state reachability or no deadlocks and starvations. On the other hand, the provision of operational semantics makes it possible to automatically derive IP implementation from protocol specification, easing its simulation and the automatic generation of proxies that assure that each participant effectively complies with the protocol rules and provides assistance for protocol machine learning.

The features of ACSL have led to the use of the concept of *Structural Operational Semantics* (SOS) [13,14] as an approach for specifying the dynamic meaning of IPs. The *dynamic meaning* of a protocol is obtained from the dynamic meaning of the different syntactic constructs that appear in its specification. It covers the execution of the specification, including expression evaluation, message sending and reception and the execution of other non-communicative actions.

The SOS denotes a formalism that can specify the meaning of a language by means of syntactic transformations of the programs or specifications written in this language. Some special points had to be taken into account to apply the SOS formalism, designed for programming languages, to a specification language such as ACSL. The definition of operational semantics suited for ACSL represents a three-step process:

1. Definition of a terminal and term-rewriting labeled transition system based on the operational semantics described in [15],
2. Definition of the interpreter $I$ for this system, as proposed in [13], whose behavior is specified by a set of production rules.
3. Process of outputting the interpreter for each ACSL construct.

The following subsections describe the process of defining the operational semantics of an ACSL specification, stressing these points.

### 4.1 Defining the Transition System

This section presents the term-rewriting transition system developed for the ACSL language, based on a proposal by [15]. To produce a *term-rewriting labeled transition system* $\langle \Gamma, \Lambda, \rightarrow, \Upsilon \rangle$ to fit ACSL, it is established that $\Gamma \subseteq \Sigma \times \Theta$.

By way of a configuration, $\gamma = \langle \sigma, \theta \rangle \in \Gamma$ is composed of an identifier of the *thread of interaction* $\sigma \in \Sigma$ and the parameter set $\theta \subseteq \Theta$ that describes the runtime context for that thread, where $\Sigma$ is the set of *threads of interaction* declared in the ACSL specification of a conversation and $\Theta$ is the set of parameters declared in those threads. In other words, a setting $\gamma$ describes a conversational state of the specified protocol, $\Sigma$ is the alphabet of conversational states and $\Theta$ is an adjustment parameter set for those states.

Similarly, the set of labels $\Lambda$ is established as a set of pairs $\langle \phi, m \rangle$ composed of the cartesian product of the finite set of message exchanges $m \in M$ occurring in the ACSL specification with the set of predicates about the environment

$\phi \in \Phi(\omega)$. Based on this, the conditions of the language constructs are expressed as *pick*, *while* or *switch* (and, therefore, the parameters are declared by means of the *paramSetDecl* construct). Some of these predicates refer to exchanged messages (appear associated with *exchange* elements in *paramSetRef* elements) and are, therefore, denoted $\phi(m)$ and associated with the exchanged messages for the purpose of specifying this relation.

The transition relation $\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ is now $\rightarrow \subseteq (\Sigma \times \Theta) \times (\Phi \times M) \times (\Sigma \times \Theta)$. The $\rightarrow$ relation therefore represents a transition relation between conversational states labeled by means of an action. The idea, as in other labeled transition systems, is that the action associated with a transition provides information about what is happening in the setting during the transition (internal actions) and/or about the interaction between the agents and their environment (external actions). In this case, the actions refer to interagent communication, and therefore the information they supply is the actual messages exchanged and the settings information (parameters of the conversational states) they use.

Accordingly, the language's alphabet is made up of the set of all possible messages exchanged in the course of a conversation $M = \{m_i, i = 1..n\}$. Hence, this language can be defined as a set of possible sequences of exchanged messages, each of which is a word of the language:

$$L \equiv \varepsilon \in L \mid m \in M \rightarrow m \in L \mid s_1, s_2 \in L' \rightarrow s_1 \odot s_2 \in L'$$

Finally, the set of term settings $\Upsilon \subseteq \Gamma$ is determined by those settings with interaction threads in which a message labeled as *term* is exchanged, as these are the only messages for which the following holds:

$$\forall \gamma \in \Upsilon, \forall \gamma' \in \Gamma \cdot \gamma \nrightarrow \gamma'$$

The transition system presented above is based on a definition given by [13]. However, the set of actions $\alpha \subseteq \Lambda(\omega)$ to be executed in each transition can be added to this definition. The transition relation would then be either:

$$\rightarrow \subseteq (\Sigma \times \Theta) \times (\Phi \times M) \times (\Sigma \times \Theta) \times \alpha \subseteq \Lambda(\omega)$$

or

$$\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma \times \Lambda \rightarrow \subseteq (\Sigma \times \Theta) \times (\Phi \times M) \times (\Sigma \times \Theta) \times (\Lambda)$$

## 4.2 Defining the Interpreter

According to [14], the SOS formalism can be used to build the operational semantics of an ACSL specification by formally describing an interpreter $I$ of that language whose behavior is specified by means of a set of production rules.

Following ideas taken from [16], $I$ is modeled as a function whose argument is an ACSL-specified protocol$P$ and an environment $\omega$, and which describes the behavior of $\langle P, \omega \rangle$ as an [in]finite series of productions like $\langle P, \omega \rangle \rightarrow \langle P_1, \omega_1 \rangle \rightarrow \langle P_2, \omega_2 \rangle \rightarrow \ldots$ If $P$ ends, then the result is $\langle END, \omega_n \rangle$.

Accordingly, the automaton specification, which acts as an interaction protocol interpreter and therefore determines the operational semantics of an ACSL specification, defines a set of production rules that constitutes the definition of the respective interpreter, whereas the sequences of messages sent and received is the program that is to be interpreted. Supposing that the agents are modeled on an internal BDI architecture, the set of beliefs ($\beta$), desires ($\delta$) and intentions ($\iota$) makes up the environment $\omega$ and predicates about that environment (e.g. WantToPropose(p), IntendToDo(t), etc.). Consequently, $\Phi(\omega) \subseteq \beta \cup \delta \cup \iota$ and the actions set (including the exchanged messages) are the lateral effects on $\omega$ in the same way as a variable is assigned in a programming language.

As mentioned earlier, the operational semantics developed is based on a production system that maps conversational states to new conversational states for a given ACSL specification. The format of a production rule is shown in the following.

This transition system can be viewed as a production system in which each transition is determined by a rule being fired when an action takes place and subject to the validity of the predicate $\phi$. Let a transition be denoted

$$\langle\langle e \in \Sigma, \theta \in \Theta\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[msg/\phi(msg)]{} \langle\langle e' \in \Sigma, \theta' \in \Theta\rangle, [\alpha]/\alpha \in \Lambda(\omega)\rangle$$

This action can represent message sending, reception or an internal agent action. Therefore, three production rule types are accounted for: (a) production rules fired by message sending, if $\phi$ is true, (b)production rules fired by message reception, if $\phi$ is true, and (c) production rule fired by an internal agent action, if $\phi$ is true.

They all qualify the transition relation with $msg/\phi(msg)$ message template sending $\xrightarrow[msg/\phi(msg)]{}$ or reception $\xrightarrow{msg/\phi(msg)}$, or with $\xrightarrow[\varepsilon]{}$ for internal actions. The $msg$ format is identical to the one used to represent a parameterized setting $\langle m \in M, \theta \in \Theta\rangle$, where $M$ is the alphabet of performatives and $\theta$ is a parameter adjustment tuple for the message.

Using the predicate $\phi(msg)$ about the messages received, it is possible, for example, to find out if the message source already sent another message earlier (the source's membership of the group of agents participating in the protocol would also have to be considered). This predicate represents the part of the predicate that appears in the premise directly related to the message.

$$\phi(msg) = true \leftrightarrow \neg\exists m \in msgQ/m.from = msg.from$$

The parameterized description of a conversational state of the protocol $\langle e \in \Sigma, \theta \in \Theta\rangle$ is a state identifier $e$ belonging to the states alphabet $\Sigma$ and a tuple $\theta$ of adjustment parameters for the state $e$. The adjustment parameters tuple includes (1) variables of type Int, Char, List, Tuple for template adjustment or (2) variables for representing beliefs, desires and intentions.

Different types of template adjustment are accounted for depending on the parameter type to which they are applied. Accordingly, the adjustments considered for the type Int are $Succ(N), N \neq 0$ and $Zero$, and the transition function $Succ(Succ(N)) \rightarrow Succ(N)$. On the other hand, the adjustments $p :: rest$ and $[]$, and the transition function $p :: rest \rightarrow rest, rest \neq []$ and $p :: [] \rightarrow []$ are considered for the type List.

$\Phi(\omega)$ represents the set of predicates about the environment that can be evaluated by the agents participating in the conversation. These predicates are usually intrinsically related to agents' beliefs, desires and intentions. However, no assumptions are made about how the agents conduct the evaluation, as this may be related not only to the protocol conversational states but also to the agents' internal state.

## 4.3 Process of Outputting the Interpreter for Each ACSL Construct

The following subsections detail the process of generating the interpreter for key ACSL language constructs. Concurrent and synchronization related ACSL constructs are left for a forthcoming paper.

**Simplifications.** The following simplification rule is used with the aim of simplifying the generation of production rules in embedded constructs:

If there are two rules

$$\langle\langle A \in \Sigma, \theta \in \Theta\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[msg/\phi(msg)]{} \langle\langle B \in \Sigma, \theta' \in \Theta\rangle, \alpha \in \Lambda(\omega)\rangle$$
$$\langle\langle B \in \Sigma, \theta' \in \Theta\rangle, \phi' \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle C \in \Sigma, \theta'' \in \Theta\rangle, \alpha' \in \Lambda(\omega)\rangle$$

they can be simplified as

$$\langle\langle A \in \Sigma, \theta \in \Theta\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[msg/\phi(msg)]{} \langle\langle C \in \Sigma, \theta'' \in \Theta\rangle, \alpha' \in \Lambda(\omega)\rangle$$

Provided that $\phi \to \phi'$. The same applies to message sending rules.

**Production Rules for Receiving n Messages.** The delay in receiving n messages from different agents is expressed by means of the pick instruction:

```
Pick := pick [Times] [ParamSetRef] {EventHandler} [OnTimes]
EvenHandler := eventHandler Event ActionBlock
OnTimes := onTimes (ThreadOfInteraction | ProtocolControlGroup | Action)
EventHandler := eventHandler Event ActionBlock
Event := event Id (DelayFor | DelayUntil | Exchange | Catch)
ActionBlock := action (ThreadOfInteraction | ProtocolControlGroup | Action)
Times := Expression
Id := id string
```

The following standard set of SOS production rules is obtained by each *EventHandler* for this instruction:

$$\langle\langle A \in \Sigma, (\ldots \mathit{Succ}(t) \ldots)\rangle, \mathit{true}\rangle \xrightarrow[msg_1/\phi(msg_1)]{} \langle\langle A, (\ldots (t) \ldots)\rangle, [\alpha \in \Lambda(\omega)]\rangle\rangle$$
$$\langle\langle A \in \Sigma, (\ldots \mathit{Succ}(t) \ldots)\rangle, \mathit{true}\rangle \xrightarrow[msg_2/\phi(msg_2)]{} \langle\langle A, (\ldots (t) \ldots)\rangle, [\alpha \in \Lambda(\omega)]\rangle\rangle$$
$$\ldots \langle\langle A \in \Sigma, (\ldots \mathit{Zero} \ldots)\rangle, \mathit{true}\rangle \xrightarrow[\varepsilon]{} \langle\langle B \in \Sigma, \{\theta \in \Theta\}\rangle, [\alpha \in \Lambda(\omega)]\rangle\rangle$$

where the expressions $(\ldots \mathit{Succ}(t) \ldots)$ include all the parameters referenced in the body of the pick (including events) and the expression $(\theta \in \Theta)$ will be composed of the set of values that instantiate the thread parameters referenced in the <onTimes> expression.

All the pick handlers have been assumed to be concerned with message delay. Otherwise, the handler delay condition would be stated in $\phi \in \Phi(\omega)$, which would no longer be *true*, and the transition rule would switch to $\xrightarrow[\varepsilon]{}$.

Accordingly, for the next use of *pick*, taken from the ACSL specification of the *FIPA IteratedContractNet* protocol [5] (the *proposer* agent gather *inform*, *failure*, and *end* messages from participants):

```
<pick times="length(apl)">
  <paramSetRef><paramRef mode="match">apl</paramRef></paramSetRef>
  <eventHandler> <event> <exchange message="Inform" direction="in" mode="middle">
                            <paramSetRef> <paramRef mode="adjust">p(t1)-->ipl</paramRef>
                                          <paramRef mode="match">t1</paramRef>
                            </paramSetRef></exchange></event>
                 <action><empty/></action></eventHandler>
  <eventHandler> <event> <exchange message="Failure" direction="in" mode="middle">
                           <paramSetRef> <paramRef mode="adjust">p(t1)-->fpl</paramRef>
                                         <paramRef mode="match">t1</paramRef>
                           </paramSetRef></exchange></event>
                 <action><empty/></action></eventHandler>
  <onTimes> <threadOfInteractionRef threadRef="End">
               <paramSetInst> <paramInst> <ref>t1</ref> <value>t1</value></paramInst>
                              <paramInst> <ref>pl</ref> <value>fpl</value></paramInst>
               </paramSetInst></threadOfInteractionRef> </onTimes> </pick>
```

we get the following SOS rules:

$$\langle\langle A \in \Sigma, (t_1, Succ(n), fpl, ipl)\rangle, true\rangle \xrightarrow[Fail(p(t_1))]{} \langle\langle A, (t_1, n, p :: fpl, ipl)\rangle, NO)\rangle$$
$$\langle\langle A \in \Sigma, (t_1, Succ(n), fpl, ipl)\rangle, true\rangle \xrightarrow[Inform(p(t_1))]{} \langle\langle A, (t_1, n, fpl, p :: ipl)\rangle, NO)\rangle$$
$$\langle\langle A \in \Sigma, (T_1, Zero, fpl, ipl)\rangle, true\rangle \xrightarrow[\varepsilon]{} \langle\langle B \in \Sigma, t_1, fpl\rangle, NO)\rangle$$

**Iteration Production Rules.** Iterations are expressed in ACSL by means of the while instruction:

```
While := while Condition ActionBlock
Condition := condition [ParamSetRef] Expression
ActionBlock := action (ThreadOfInteraction | ProtocolControlGroup | Action)
```

for which the following set of standard SOS production rules is obtained:

$$\langle\langle A \in \Sigma, \theta \in \Theta\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle A \in \Sigma, \theta_1 \in \Theta\rangle, \alpha \in \Lambda(\omega)\rangle$$
$$\langle\langle A \in \Sigma, \theta_2 \in \Theta\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle B \in \Sigma, \theta_3 \in \Theta\rangle, \alpha \in \Lambda(\omega)\rangle$$

When the while instruction ends, $\Theta$ converges to a state in which $\theta_2$ is true.

The following example assumes that a message is to be sent to all the agents identified in a list:

```
<threadOfInteraction>
 <while> <condition condition="existProposalInProposals">
            <paramSetRef> <paramRef mode="adjust">p(t1)::pl</paramRef>
                          <paramRef mode="match">t1</paramRef></paramSetRef> </condition>
         <action> <exchange message="msg" direction="out" delivery="unreliable"
                                           mode="middle" type="asynchronous">
                   <paramSetRef> <paramRef mode="match">p.from</paramRef>
                                 <paramRef mode="match">p</paramRef></paramSetRef>
       </exchange></action></while> </threadOfInteraction>
```

for which the following set of SOS rules is produced:

$$\langle\langle A \in \Sigma, \ldots p :: l \ldots\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[msg(p.from)]{} \langle\langle A \in \Sigma, \ldots l \ldots\rangle, \alpha \in \Lambda(\omega)\rangle$$
$$\langle\langle A \in \Sigma, \ldots [] \ldots\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle B \in \Sigma, \theta \in \Theta\rangle, \alpha \in \Lambda(\omega)\rangle$$

In this case, the while instruction is guaranteed to end, since $\Theta \equiv p :: l$ y $\theta_1 \equiv l$, which necessarily has $\Theta$ converge to $[]$, making $\theta_2$ true.

**Optionality Production Rules.** ACSL can be used to express optionality in the course of a conversation by means of the *switch* construct. The overall structure of this construct is shown below:

```
Switch := switch Multichoice {Branch} [Default]
Multichoice := multichoice boolean
Branch := branch Case ActionBlock
Default := default (ThreadOfInteraction | ProcolControlGroup | Action)
Case := case Condition [ParamSetRef]
Condition := condition [ParamSetRef] Expression
ActionBlock := action (ThreadOfInteraction | ProtocolControlGroup | Action)
ParamSetRef := paramSetRef {ParamRef}
ParamRef := paramRef Mode string
Mode := match | adjust
```

The following rule template is obtained for each *branch*:

$$\langle\langle A \in \Sigma, \{\theta_i \in \Theta\}\rangle, \phi \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle B_j \in \Sigma, \{\theta'_k \in \Theta\}\rangle, [\alpha \in \Lambda(\omega)]\rangle$$

where $A \in \Sigma$ denotes the conversational state generated for the switch instruction, $\{\theta_i \in \Theta\}$ is the list of referenced parameters (*paramSetRef*) in the respective branch condition, $\phi \in \Phi(\omega)$ is the actual condition, $B_j \in \Sigma$ denotes another conversational state that will be used in the antecedent of the rules generated for the interaction thread defining the action of this branch. If this is a reference to an interaction thread, $\{\theta'_k \in \Theta\}$ will be the set of values that instantiate the parameters of the respective thread (*paramSetInst*).

The same rule template is obtained for the *default* branch considering:

$$\phi = \bigcup_{i=1}^{n} \phi_i \qquad \text{and} \qquad \theta = \neg \bigvee_{i=1}^{n} \theta_i$$

The next section gives an example of a set of optionality and iteration rules.

**Structure Composition Rules.** The rules resulting from applying the templates discussed in earlier sections (including the simplification template) to an ACSL specification are shown below. This ACSL specification (excerpt) is made up of a *while* structure plus a *switch* structure. The fragment, taken from the ACSL specification of the *FIPA IteratedContractNet* protocol, models how the proposer, after receiving the proposals from the contract net, notify to each agent having sent a proposal if its proposal has been accepted (*Accept* message) or rejected (*Reject* message):

```
<threadOfInteraction>
 <while> <condition condition="existProposalInProposals">
       <paramSetRef> <paramRef mode="adjust">p(t1)::pl</paramRef>
                     <paramRef mode="match">t1</paramRef> </paramSetRef></condition>
       <action> <switch multiChoice="false">
                <branch> <case condition="WantToAcceptProposal">
                          <paramSetRef> <paramRef mode="match">p</paramRef>
                                        <paramRef mode="adjust">p-->apl</paramRef>
                          </paramSetRef></case>
                         <action> <exchange message="Accept" direction="out" mode="middle"
                                           delivery="unreliable" type="asynchronous">
                                    <paramSetRef> <paramRef mode="match">p.id</paramRef>
                                                  <paramRef mode="match">p</paramRef>
                                    </paramSetRef></exchange></action></branch>
```

```
<branch> <case condition="WantToRejectProposal">
          <paramSetRef> <paramRef mode="match">p</paramRef>
          </paramSetRef></case>
         <action> <exchange message="Reject" direction="out" mode="middle"
                            delivery="unreliable" type="asynchronous">
                  <paramSetRef> <paramRef mode="match">p.id</paramRef>
                                <paramRef mode="match">p</paramRef>
                  </paramSetRef></exchange></action></branch>
</switch> </action> </while> </threadOfInteraction>
```

Firstly, before any simplifications are made, we have:

$$\langle\langle \mathrm{A}, t_1, p(t_1) :: pl, apl\rangle, true\rangle \xrightarrow[\varepsilon]{} \langle\langle \mathrm{B}, t_1, pl, p, apl\rangle, \mathrm{NO}\rangle$$
$$\langle\langle \mathrm{B}, t_1, pl, p, apl\rangle, WantToAcceptProposal(p)\rangle \xrightarrow{Accept(p)} \langle\langle \mathrm{A}, t_1, pl, p :: apl\rangle, NO\rangle$$
$$\langle\langle \mathrm{B}, t_1, pl, p, apl\rangle, WantToRejectProposal(p)\rangle \xrightarrow{Reject(p)} \langle\langle \mathrm{A}, t_1, pl, apl\rangle, NO\rangle$$
$$\langle\langle \mathrm{A}, t_1, [], apl\rangle, true\rangle \xrightarrow[\varepsilon]{} \langle\langle \mathrm{C}, l\rangle, \mathrm{NO}\rangle$$

which, simplified (according to ), becomes:

$$\langle\langle \mathrm{A}, t_1, p(t_1) :: pl, apl\rangle, WantToAcceptProposal(p))\rangle \xrightarrow{Accept(p)} \langle\langle \mathrm{A}, t_1, pl, p :: apl\rangle, NO\rangle$$
$$\langle\langle \mathrm{A}, t_1, p(t_1) :: pl, apl\rangle, WantToRejectProposal(p)\rangle \xrightarrow{Reject(p)} \langle\langle \mathrm{A}, t_1, pl, apl\rangle, NO\rangle$$
$$\langle\langle \mathrm{A}, t_1, [], apl\rangle, true\rangle \xrightarrow[\varepsilon]{} \langle\langle \mathrm{C}, l\rangle, \mathrm{NO}\rangle$$

**Sequential Statement Composition Rules.** The transition for the sequential statement composition $s_1; S$ is derived from the transition for the statement $s_1$.

$$\frac{\begin{array}{l}\langle\langle A \in \Sigma, s_1, \theta_1 \in \Theta\rangle, \phi_1 \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle A_1 \in \Sigma, \phi, \theta_1' \in \Theta\rangle, \alpha_1 \in \Lambda(\omega)\rangle \ \ldots \\ \langle\langle A \in \Sigma, s_1, \theta_2 \in \Theta\rangle, \phi_2 \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle A_2 \in \Sigma, \phi, \theta_2' \in \Theta\rangle, \alpha_2 \in \Lambda(\omega)\rangle\end{array}}{\begin{array}{l}\langle\langle A \in \Sigma, s_1; S, \theta_1 \in \Theta\rangle, \phi_1 \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle A_1 \in \Sigma, S, \theta_1' \in \Theta\rangle, \alpha_1 \in \Lambda(\omega)\rangle \ \ldots \\ \langle\langle A \in \Sigma, s_1; S, \theta_2 \in \Theta\rangle, \phi_2 \in \Phi(\omega)\rangle \xrightarrow[\varepsilon]{} \langle\langle A_2 \in \Sigma, S, \theta_2' \in \Theta\rangle, \alpha_2 \in \Lambda(\omega)\rangle\end{array}}$$

The DIALOG project Web site [12] contains, both for reference and for better understanding of the paper, examples of complete $AUML^+$ diagrams, ACSL specifications, and SOS interpreters for a number of relevant IPs.

## 5 Conclusions

In this paper, we have stressed that the problem with existing approaches to agent interaction modeling is that there is a huge void between the proposals based on formal techniques, whose design remains extremely complex, and the graphic notation-based techniques, which are devoid of precise semantics and rule out automatic specification exchange and interpretation for the purpose of specification simulation, validation and execution.

Bearing this in mind, we have presented the rationale behind DIALOG: a formal framework that considers all the stages of a protocol engineering process, i.e. the design, specification, validation, implementation and management of IPs, thanks to the three views into which it is organized. The paper has focused on the developed SOS and has highlighted how this formal semantics allows protocol property verification and eases automatic rule-based code generation from an

ACSL specification for the purpose of simulating IP code execution at design time, as well as improving and assuring correct IP compliance at run time. We have also stressed throughout the paper how the availability of a syntactic specification of an IP in a declarative-type machine-readable language such as the proposed ACSL helps to improve IP publication, discovery and communication on the Web, as well as the machine learning of IP by agents.

## References

1. Finin, T; Labrou, Y.; and Mayfield, J. KQML as an agent communication language. In J. M. Bradshaw (Ed.) Software Agents. MIT Press (1997)
2. Foundation for Intelligent Physical Agents. FIPA ACL message representation in string specification. http://www.fipa.org/specs/fipa00070/ (2000)
3. McBurney, P., Parsons, S., and Wooldridge, M. Desiderata for Agent Argumentation Protocols. In Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS02), Bologna, Italy (2002)
4. Greaves, M.; Holmback, H.; and Bradshaw, J. What is a conversation policy?. In F. Dignum and M. Greaves (Eds.) Issues in Agent Communication, volume 1916 of Lecture Notes in Artificial Intelligence, pages 118–131. Springer (2000)
5. Foundation for Intelligent Physical Agents. FIPA Interaction protocol Library Specification. http://www.fipa.org/specs/fipa00025, FIPA (2001)
6. Hanachi, C., Sibertin-blanc, C.: Protocol Moderators as Active Middle-Agents in Multi-Agent Systems. Autonomous Agents and Multi-Agent Systems, 8, 131-164, Kluwer Academic Publishers, The Netherlands (2004)
7. Dignum, F, Greaves, M (eds.): Issues in Agent Communication. LNAI 1916 State-of-the-Art Survery , Springer, Heidelberg (2000)
8. Gutnik, G. and Kaminka, G.A. Representing Conversations for Scalable Overhearing, Journal of Artificial Intelligence Research, Volume 25, pages 349–387 (2006)
9. Odell J. et al. Representing agent interaction protocols in UML. In Proceedings of 1st International Workshop on Agent-Oriented Software Engineering, Limerick, Ireland (2000)
10. Alonso, F; Frutos, S; López, G; and Soriano, J. A Formal Framework for Interaction Protocol Engineering, LNAI, vol. 3690, pp. 21-30, Springer-Verlag: Berlin (2005)
11. Soriano, J; Alonso, F; and López, G. A Formal Specification Language for Agent Con-versations, LNAI, vol. 2691, pp. 214-225, Springer-Verlag: Heidelberg, (2003)
12. DIALOG Project Web Site. Computer Networks & Web Technologies Lab. Available at http://hydra.ls.fi.upm.es/research/conwetlab
13. Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI FN-19. Aarhus University, Computer Science Department, Denmark (1981)
14. Hennessy, M.: The Semantics of Programming Languages: An Introduction Using Structured Operational Semantics. Wiley (1990)
15. R. van Eijk, F. de Boer, W. van der Hoek and J-Ch. Meyer: Operational Semantics for Agent Communication Languages. In F. Dignum and M. Greaves (eds.) Issues in Agent Communication, LNCS 1916, 80-95, Springer, Heidelberg (2000)
16. Koning J.; and Oudeyer, P. Introduction to POS: Protocol Operational Semantics. International Journal of Cooperative Information Systems, 10(2):101–123 (2001)
17. Haddadi, A.: Communication and Cooperation in Agent Systems: A Pragmatic Theory. volume 1056 of LNCS. Springer Verlag, Heidelberg, Germany (1996)