Chapter 4

CLAIM AND SYMPA: A PROGRAMMING ENVIRONMENT FOR INTELLIGENT AND MOBILE AGENTS

Amal El Fallah Seghrouchni and Alexandru Suna

LIP6 - CNRS UMR 7606 University of Paris 6
8, Rue du Capitaine Scott
75015, Paris
{Amal.Elfallah,Alexandru.Suna}@lip6.fr

Abstract The multi-agent systems (MAS) paradigm is one of the most important and promising approaches to occur in computer science during the 90s. However, for an effective use of the agent technology in real life applications, specific programming languages are required. CLAIM is a high-level agent-oriented programming language that combines cognitive aspects such as knowledge, goals and capabilities and computational elements such as communication, mobility and concurrence in order to reduce the gap between the design and the implementation phase. CLAIM has an operational semantics that is a first step towards the verification of the built MAS. The language is supported by a distributed platform called SyMPA, implemented in Java, compliant with the specifications of the MASIF standard from the OMG, that offers all the necessary mechanisms for a secure execution of a distributed MAS. CLAIM and SyMPA have been used for developing several applications that proved the expressiveness of the language and the robustness of the platform.

Keywords: Agent-oriented programming, mobile agents, ambient calculus.

4.1 Motivation

The emergence of autonomous agents and multi-agent technology is one of the most exciting and important events to occur in computer science during the 1990s. The main focus of the multi-agent systems (MAS) community has been on the development of informal and formal tools (*e.g.* consortiums such as FIPA[1] or OMG[2] have attempted to propose a wide range of standards to cover the main aspects of MAS engineering), concepts (*e.g.* concerning mental or social attitudes, communication, co-operation, organization), techniques (*e.g.* AUML[3]) and modal languages (*e.g.* BDI[182]) in order to be able to analyze and specify MAS. Unfortunately, the design of declarative languages and tools which can effectively support MAS programming and allow implementing the key concepts of MAS remained at an embryonic stage. In addition, the potential of MAS technology for large-scale, cross-functional deployment of general purpose in industrial setting has been hampered by insufficient progress on infrastructure, architecture, security and scalability issues.

Recently, the mobile agents technology (the mobility is seen as a transversal property for agents) tries to improve the systems' performances since it provides powerful programming constructs for designing distributed and mobile applications. Thanks to the mobile agents paradigm, it becomes easy to design *active entities* that move over the network and perform tasks on hosts (*target sites* or *computers*), thus reducing the network traffic and increasing the scalability and the flexibility of such applications.

Despite the plethora of approaches and platforms that have been proposed for mobile agents, the main focus remains on the development of mobile objects and processes. Mainly implemented using object-oriented frameworks, the mobile agents provide a collection of extensible classes modelling simple concepts of agent that are specified rather at the implementation level.

For an effective use of the MAS paradigm, we claim that specific high-level programming languages are required. The programming environment presented in this chapter is motivated by three main objectives:

1. Propose an agent oriented programming language that:

   - helps the designer to reduce the gap between the design and the implementation phases; *i.e.* the designer should think and implement using the same paradigm, namely through agents;
   - allows the representation of cognitive skills such as knowledge, beliefs, goals and more complex mechanisms such as planning, decision making and reasoning;
   - meets the requirements of mobile computation in order to support the geographic distribution of complex systems and of their computation over the net;

[1]FIPA on-line : http://www.fipa.org
[2]OMG : http://www.omg.org
[3]AUML : http://www.auml.org/

- allows the dynamic adaptability and reconfiguring of the MAS. Thanks to mobility, to the hierarchical representation of agents and to the language' features, our agents (and consequently the MAS) are able to reconfigure themselves autonomously, to acquire new knowledge and capabilities and to dynamically adapt their structure in accordance with the changes in the environment and the demands of target applications.

2. Make possible the verification of MAS. Indeed, at a short term we would like an agent-oriented programming language that allows the verification of the built systems. A first and necessary step towards developing methods for verifying formally agent-oriented programs is the design of a suitable operational semantics. It opens the way to the application of standard techniques like type systems or model-checking to the setting of agent-oriented programming.

3. Provide a distributed platform that supports the proposed language and the deployment and secure execution of mobile MAS.

To reach our objectives, we proposed a high-level declarative language called CLAIM (Computational Language for Autonomous, Intelligent and Mobile agents) [81] that combines the main advantages of the intelligent agents paradigm (*e.g.* intelligence, autonomy, communication primitives and cognitive skills) with those of the concurrent languages such as the ambient calculus [41] (*e.g.* concurrence, hierarchical representation of agents and mobility primitives). CLAIM has an operational semantics [83] that is a first step towards the verification of the built MAS. The language is supported by a distributed platform, called SyMPA (SYstem Multi-Platform of Agents) [211] that offers all the necessary mechanisms for the deployment of distributed MAS designed in CLAIM and for its secure execution.

4.2 Language

CLAIM is a high-level declarative language allowing to design intelligent and mobile agents.

4.2.1 Specifications and Syntactical Aspects

A MAS in CLAIM is a set of hierarchies of agents distributed over a network. The notion of hierarchy in our approach can be also seen as a membership relation. Thus, "an agent is sub-agent of another agent" means that he is contained in the higher-level agent. A CLAIM agent is a node in a hierarchy; he is an autonomous, intelligent and mobile entity that can be seen as a bounded place where the computation happens and has a parent, a list of local processes and a list of sub-agents. In addition, an agent has intelligent components such as knowledge, capabilities, goals, that allow a reactive or proactive behavior.

In CLAIM, agents and classes of agent can be defined using:

```
defineAgent agentName {
    authority=null; | agentName ;
    parent=null; | agentName ;
    knowledge=null; | { (knowledge;)+}
    goals=null; | { (goal;)+}
    messages=null; | { (queueMessage;)+}
    capabilities=null; | { (capability;)+}
    processes=null; | { (process |)* process }
    agents=null; | { (agentName;)+}
}
defineAgentClass className ( (arg,)*) {...}
```

An new agent can be instantiated from an already defined class using the primitive:

```
newAgent name:className ( (arg,)*)
```

In CLAIM, variables (denoted by *?x*) can be used to replace agents' names, messages, goals, etc. There are global (for a class) or local (to a capability) variables. The agents' components we propose allow representing the agents' mental state, communication and mobility and will be presented below. Most of the components are *null* in the definition (*e.g.* parent, messages, etc.) but will evolve during the agent's execution.

An agent is uniquely identified in the MAS by his name and he belongs to an authority. Thus, the **authority** component is instantiated at the agent's creation and is composed of the authority and the name of the agent that has created the current agent. This component is necessary for security reasons (*e.g.* for authentication).

The agents in CLAIM are hierarchically represented, like the *ambients* [41]. So an agent's **parent** is represented by the name of the agent that currently contains him. When an agent is created, his parent and his authority indicate the same agent; after the migration, his parent will change, but his authority will always be the same.

The **knowledge** component contains pieces of information about other agents (*i.e.* about theirs capabilities or their classes) or about the world (divers propositions). This knowledge base is a set of elements of *knowledge* type, defined as follows:

```
knowledge ::=  agentName(capabilityName,message,effect)
            |  agentName:className
            |  proposition
```

We can notice that the knowledge about other agents has a standard format, containing the name of the known agent and his class or capability. In addition, the user can define his own ontology of information about the world, represented as propositions containing a name and a list of arguments.

$proposition = name(arg_1, arg_2, ..., arg_n)$

Propositions can also be used for denoting goals or messages.

The current **goals** of an agent are represented as user-defined propositions, in accordance with the current application. The agent will try to achieve his goals using his capabilities or services offered by other agents.

The CLAIM agents communicate asynchronously using messages. Every agent has a queue for storing the received **messages**. The messages are processed using a FIFO policy and are used to activate capabilities. A message from the queue contains the sender of the message and the arrived message:

queueMessage ::= agentName > message

An agent can send messages to an agent (*unicast*), to all the agents in a class (*multicast*), or to all the agents in the system (*broadcast*), using the primitive:

send(receiver,message), where the receiver can be:

- ***this*** - the message is sent to himself;

- ***parent*** or ***authority*** - the message is sent to the agent's current parent or authority (the agent that created the current agent);

- *agentName* - the message is sent to the specified agent;

- ***all*** - the message is sent to all the running agents;

- *?Ag:className* - the message is sent to all the agents that have been instantiated from the specified class of agents;

In CLAIM there are three types of messages:

1. *propositions*, defined by the designer to suit the current application and used to activate capabilities;
2. the **messages concerning the knowledge**, used by agents to exchange information about their knowledge and capabilities. These messages have a predefined treatment, but a designer can write capabilities to treat them in a different manner:

- ***tell(knowledge)*** - to give an agent a piece of information; the specified knowledge is added in the agent's knowledge base.
- ***askAllCapabilities()*** - an agent requests all the capabilities of another agent; The later inform the first agent about all his capabilities, using the **tell** primitive.
- ***askIfCapability(capabilityName)*** - an agent asks another agent if he has the specified capability; If the later has this capability, he confirms using the **tell** communication primitive.

- ***achieveCapability****(capabilityName)* - an agent requests from another agent the execution of the specified capability; if this capability's condition is verified, it is executed.
- ***askEffect****(effect)* - to ask the achievement of an effect from another agent.
- ***doneEffect****(effect)* - to confirm the accomplishment of an effect.

3. the **mobility messages** are used by the system during the mobility operations, for asking, granting or not granting mobility permissions. Their treatment can be redefined by the designer in order to control the mobility. They are represented at the semantical level by co-actions. In the *ambient calculus*, the only condition for the mobility operations is a structure condition (*e.g.* for the *enter* operation, the involved agents must be on the same level in the agents' hierarchy). In CLAIM, we kept this condition, but we added the mobility messages for an advanced security and control.

The **capabilities** are the main elements of an agent and dictate his behavior. They represent the actions an agent can do in order to achieve his goals or that he can offer to other agents. A *capability* has a message of activation, a condition, the process to execute in case of activation and a set of possible effects:

```
capability ::= capabilityName {
    message=null; | message;
    condition=null; | condition;
    do {process}
    effects=null; | {(effect;)+ }
}
```

To execute a capability, the agent must receive the activation message and verify the condition. If the message is ***null***, the capability is executed whenever the condition is verified. If the condition is ***null***, the capability is executed when the message is received. A condition can be a Java function that returns a *boolean*, an achieved effect, a condition about agent's knowledge or sub-agents, or a logical formula:

```
condition ::=  Java(objectName.function(args))
             | agentName.effect
             | hasKnowledge( knowledge )
             | hasAgent( agentName )
             | not( condition )
             | and( condition,(condition)+ )
             | or( condition,(condition)+ )
```

An agent concurrently executes several **processes**. One of these concurrent processes can be a sequence of processes, an instruction, a variable's instantiation, a method implemented in other programming language (*e.g.* Java), the invocation of a known Web Service, the creation of a new agent or the removal on an existing one, a mobility operation or a message transmission:

```
process ::=  process.process
          |  instruction
          |  ?x = (value | Java(obj.method(args)))
          |  Java(obj.method(args))
          |  WebService(address,method(args))
          |  newAgent agentName:className( (arg,)*)
          |  kill (agentName)
          |  open (agentName)
          |  acid
          |  in (mobilityArgument,agentName)
          |  out (mobilityArgument,agentName)
          |  move (mobilityArgument,agentName)
          |  send (receiver,message)
```

We defined two instructions:

forAllKnowledge(knowledge) { process } - execute the process for all agent's knowledge that satisfy a criteria (*e.g.* all agent's knowledge about a certain agent).

forAllAgents(agentName) { process } - execute the process for all the agent's sub-agents that satisfy a criteria (*e.g.* all the agent's sub-agents that belong to a certain class).

The mobility primitives have the same utilization as in the ambient calculus but they have been adapted to intelligent agents. Hence, an agent can open the borders of one of his sub-agents (***open***) or can open his own borders (***acid***); in both cases, the parent of the open agent inherits knowledge, capabilities, processes and sub-agents from the open agent. Also, an agent can enter an agent form the same level in the hierarchy, *i.e.* having the same parent (***in***), can exit the current parent (***out***) or can migrate into another agent (***move***). With respect to the hierarchical representation of agents, these operations allow flexible reconfiguring of MAS and dynamic gathering of capabilities and knowledge.

An important problem is the migration's granularity, and the question is "who can migrate?". We specify this using the mobility argument that allows the migration of the agent himself, of a clone of the agent or of a process:

mobilityArgument = ***this*** | ***clone*** | *process*

The **agent** component represents the agent's current sub-agents.

The CLAIM language offers to the agents' designer the possibility to define two types of behavior for the agents:

The ***reactive behavior*** (or forward reasoning):

- get a message from the queue (the first or using a selection heuristic);
- find the capabilities that have this message of activation and replace the variables in the body of the capability;
- verify the conditions of the chosen capabilities;
- execute the process of the verified capabilities; let us note that several capabilities can be concurrently activated.

The ***pro-active behavior*** (or backward reasoning):

- get a goal from the list of goals (the first or using a selection heuristic);
- find the capabilities that allow to achieve this goal;
- verify the conditions of the chosen capabilities; if the condition is an agent's effect, add this effect in his list of goals; if the condition is other agent's effect, request the execution of the corresponding capability;
- execute the process of the verified capabilities.

Before reading the next section, about CLAIM's semantics, the reader can see in section 4.4 a list of applications implemented in CLAIM, one of them presented in details in order to illustrate the language's specifications.

4.2.2 Semantics and Verification

The specifications of the CLAIM programming language, presented in the previous section, are used by the programmer to define agents and classes of agents. Nevertheless, these specifications are complex and the reduction rules of the semantics using the same notations are difficult to read and understand. That's why we are using another formalism (equivalent with the specifications) to re-write the syntax and the operational semantics of the language, semantics that must take into account the mobility, the communication and the specificity of cognitive agents. All the components presented at the specification level will be also represented at the semantical level, with a different notation, to facilitate the understanding and the readability of the reduction rules.

A MAS in CLAIM is a set of connected hierarchies of agents. At the semantical level, a MAS (or a CLAIM program) is a set Π of running agents (deployed on several sites).
We consider that α, β, π, ... are agents' names. We also consider that a_1, a_2, ... are agents (with all the components) belonging to Π. The goals, the messages, the capabilities' effects and the pieces of information about the world are propositions containing a name and a list (possibly empty) of arguments,

denoted by: $\rho = n(t_1, t_2, ..., t_m)$. The other notations will be explained as they are introduced.

A program is: $\Pi = a_1 \parallel a_2 \parallel ... \parallel a_n, n \geq 0$. The notation $\parallel$ represents concurrent agents inside the MAS, running on the same computer or on different connected computers.

An agent: $a_i = \langle \alpha, \pi, K, G, G', M, C, P, S, E \rangle$, where:

- α is the agent's name;

- π is the name of the agent's current parent;

- K is the knowledge base, containing pieces of information about the world (represented as propositions) or about other agents' capabilities (containing the name, the message and the effect) or classes.

$K = \{k_1, k_2, ..., k_n\}, k_i = \rho_i \mid \alpha_i(n_i, m_i, E_i) \mid \alpha_i : cl_i$

- G is the agent's set of current goals (not treated yet); this list can contain not only agent's goals, but also goals requested by other agents, denoted by *e.g.* $\beta.g$

- G' is the agent's set of currently processing goals;

- M is the messages queue containing a set of pairs representing the sender and the message. The received messages are treated sequentially:

$M = \oslash \mid \alpha_1\{m_1\}.\alpha_2\{m_2\}.... ;$

- C is the agent's list of capabilities. A capability has a name (n_i) and triggers a process (p_i) according to a message (m_i) if a (optional) pre-condition (Ω_i) is verified. A capability may have eventual effects (post-conditions) (E_i):

$c_i = \langle n_i, m_i, \Omega_i, p_i, E_i \rangle \in C$

A condition can be a Java method that returns a boolean, an effect (used for the goal-driven behavior), a condition about the agent's knowledge, sub-agents or effects, or a logical formula. We defined a function $V : (\Omega, \Pi) \rightarrow \{true, false\}$ (detailed later), that evaluates the boolean value of a CLAIM agent condition in the context of a running MAS.

- P is the list of the agent' concurrent running processes (the notation | represents concurrent processes inside an agent): $P ::= p_i \mid p_j \mid ... \mid p_k$

- S is the set of names of the agent's sub-agents;

- E is the list of achieved goals or effects.

A process may be executed either if it is explicitly coded in the agent or as a result of a triggered capability or in order to achieve a goal. Several processes can be concurrently executed by an agent. One of these concurrent processes can be, as seen in the previous section, a (possibly empty) sequence of processes, a message transmission, the creation of a new agent (belonging to an already defined class) or the removal of an existing one, a mobility operation (we added co-actions, represented in the previous section as mobility messages), an effect achievement, a variable instantiation or an instruction (the last two). We do not treat at the semantical level the Java methods and the Web Services invocations.

$$
\begin{aligned}
p_i ::= \;& \oslash \mid p_j.p_k \mid send(\alpha, m) \mid \\
& newAgent\langle \alpha, \oslash, K, G, \oslash, \oslash, C, P, \oslash, \oslash \rangle \mid \\
& kill(\beta) \mid \\
& in(\beta) \mid \overline{in}(\alpha) \mid \\
& out(\beta) \mid \overline{out}(\alpha) \mid \\
& move(\beta) \mid \\
& open(\beta) \mid \overline{open}(\alpha) \mid \\
& acid \mid \overline{acid}(\beta) \mid \\
& addEffect(e_i) \mid \\
& ?x = value \mid \\
& forAllKnowkedge(k)\{p_j\} \mid \\
& forAllAgents(\alpha_i)\{p_j\}
\end{aligned}
$$

Additional notations

Propositions are important notions in our language. A proposition has a name and contains a set of arguments: $\rho = n(t_1, t_2, ..., t_m)$. They are used to represent goals, messages, information about the world and effects. The propositions may contain variables (denoted by $?x$) as arguments. We say that a proposition is *instantiated* if it contains no variables (all the arguments are instantiated).

DEFINITION 4.1 *A proposition* $\rho = n(t_1, t_2, ..., t_m)$ *is* equal *with another proposition* $\rho' = n'(t'_1, t'_2, ..., t'_o)$ *(notation* $\rho = \rho'$*) if* ρ *and* ρ' *are instantiated and* $n = n'$, $m = o$ *and* $\forall i \in \{1, ..., m\}, t_i = t'_i$.

DEFINITION 4.2 *A proposition* ρ belongs *to a set* Λ *(e.g.* G, E*) of propositions (notation* $\rho \in \Lambda$*) if* $\exists \rho' \in \Lambda$ *and* $\rho = \rho'$.

DEFINITION 4.3 *A proposition* $\rho = n(t_1, t_2, ..., t_m)$ corresponds *to another proposition* $\rho' = n'(t'_1, t'_2, ..., t'_o)$ *(notation* $\rho \cong \rho'$*) if* ρ' *is instantiated and* $n = n'$, $m = o$ *and* $\forall i \in \{1, ..., m\}, t_i = t'_i$ *or* t_i *is a variable.*

DEFINITION 4.4 *A proposition* ρ has a correspondent *in a set* Λ *(e.g.* G, E*) of propositions (notation* $\rho \sim\in \Lambda$*) if* $\exists \rho' \in \Lambda$ *and* $\rho \cong \rho'$.

These definitions also apply to all types of knowledge, with slight differences and with the same notations.

Conditions

The function V (as seen before, $V : conditions \rightarrow \{true, false\}$) evaluates the boolean value of a capability condition in the context of a running MAS. We will use the notation $V(\Omega)$.

$$V(null) = true$$
$$V(Java(Obj.func)) = \begin{cases} true & \text{if Java returns true} \\ false & \text{else} \end{cases}$$
$$V(this.e_k) = \begin{cases} true & \text{if } e_k \sim\in E \\ false & \text{else} \end{cases}$$
$$V(\beta.e_k) = \begin{cases} true & \text{if } \exists a_j = \langle \beta, ..., E_j \rangle \in \Pi \\ & \text{and } e_k \sim\in E_j \\ false & \text{else} \end{cases}$$
$$V(hasKnowkedge(k)) = \begin{cases} true & \text{if } k \sim\in K \\ false & \text{else} \end{cases}$$
$$V(hasAgent(\beta)) = \begin{cases} true & \text{if } \beta \in S \\ false & \text{else} \end{cases}$$
$$V(not(\Omega)) = \neg(V(\Omega))$$
$$V(and(\Omega_1, \Omega_2, ..., \Omega_m)) = V(\Omega_1) \wedge V(\Omega_2) \wedge ... \wedge V(\Omega_m)$$
$$V(or(\Omega_1, \Omega_2, ..., \Omega_m)) = V(\Omega_1) \vee V(\Omega_2) \vee ... \vee V(\Omega_m)$$

Reduction rules

We recall that a program in CLAIM contains a set of concurrent running agents: $\Pi = a_1 \parallel a_2 \parallel ...a_n$, where the notation $\parallel$ represents concurrent running agents in the MAS. For representing the semantics of CLAIM programs we choose an operational approach [169] consisting in a transition relation $\longrightarrow$ between states of a program. We use a different notation, giving a set of reduction rules, from an initial state of a program, verifying certain conditions, to another stable state, after the execution of actions by agents in the program: $\frac{\Pi}{\Pi'}$ (instead of $\Pi \longrightarrow \Pi'$).

For readability reasons we omit the unchanged components of agents. All the actions are considered to be atomic. At each step of an agent's execution, either a message is treated via a capability or a running process is executed or a goal is processed.

Terminal configuration

A very important notion for studying a program behavior is the terminal configuration. We give two related definitions, appropriate for CLAIM programs. The first one defines the termination of a CLAIM program, using the second definition that defines the termination of a CLAIM agent.

DEFINITION 4.5 (PROGRAM TERMINATION) *A CLAIM program is in a terminal configuration (denoted by Π_t) if it contains no agents (i.e. $\Pi_t = \oslash$) or if all its agents are in terminal configurations (see next definition).*

DEFINITION 4.6 (AGENT TERMINATION) *A CLAIM agent is in a terminal configuration if he has no message or goal to treat and no running process.*

Ex. $a_i = \langle \alpha, \pi, K, \oslash, \oslash, \oslash, C, \oslash, S, E \rangle$
Even if an agent can still receive messages that activate capabilities, we call this kind of configuration a terminal configuration.

Message transmission

Using the *send* primitive and the language's possibilities, an agent can send a message to himself or to another agent, to all the agents belonging to a class (multicast), or to all the agents in the MAS. The message is added at the end of the messages queue M (rule 4.1).

$$\langle \alpha, send(\beta, m) \rangle \quad \| \quad \langle \beta, M \rangle \quad \rightarrow \quad \langle \alpha, \oslash \rangle \quad \| \quad \langle \beta, M.\alpha\{m\} \rangle \tag{4.1}$$

Message processing

The arrived messages are processed sequentially, following a FIFO policy. The language offers to the designer the possibility to create his own messages, or to use several pre-defined messages (*e.g. tell, askIfCapability, askAllCapabilities, achieveCapability, askEffect, doneEffect*), that can be used by agents to exchange information about their capabilities, effects and knowledge base. These messages have a pre-defined treatment. We present next (rule 4.2) the treatment of the *tell* message, used by an agent to send a piece of information to another agent. By default, the information is added in the knowledge base. Nevertheless, the agent's designer can write a capability having this message of activation, for treating it someway else (*e.g.* verifying the trust level of the sender).

$$\langle \beta, K, \alpha\{tell(k)\} \rangle \quad \rightarrow \quad \langle \beta, K \cup \{k\}, \oslash \rangle \tag{4.2}$$

If the triggering message of a capability arrives and its condition is verified, the associated processes are executed and the effects are updated (rule 4.3). When a message arrives, the variables in the condition, process or effects are replaced with the corresponding values sent in the message. In the next reduction rule (4.3), we consider that if the capability's message m_i has a list of variables-attributes instantiated with real values in the received message, and if Ω_i, p_i and $e_i \in E_i$ contains as attributes some of the variables x_k from m_i, then Ω'_i, p'_i and $e'_i \in E_i$ will have the variables replaced with the corresponding values from the received message.

$$\frac{\langle \beta, \alpha\{m\}, C, \oslash \rangle, \text{ and } \exists \langle n_i, m_i, \Omega_i, p_i, E_i \rangle \in C, m_i \cong m \text{ and } V(\Omega'_i) = true}{\langle \beta, \oslash, C, p'_i.addEffect(e'_1)....addEffect(e'_j) \rangle, \ e'_1...e'_j \in E'_i} \tag{4.3}$$

If there are several capabilities activated by a message, the rule above is applied concurrently for each of these capabilities.

A message that does not have a corresponding capability or whose condition is not verified is simply removed from the queue, without any change in the agent's state.

Capabilities without messages

The CLAIM language gives the possibility to the agents to have capabilities that are not started by a received message, but only by a condition (*e.g.* concerning the internal state, a certain moment in time, etc.). If a capability does not have a message, it is executed whenever the condition is verified (rule 4.4).

$$\frac{\langle \beta, C, \oslash \rangle,\ \text{and}\ \exists \langle n_i, \oslash, \Omega_i, p_i, E_i \rangle \in C, V(\Omega_i) = true}{\langle \beta, C, p_i.addEffect(e_1)....addEffect(e_j) \rangle,\ e_1...e_j \in E_i} \tag{4.4}$$

Agents' creation and removal

When an agent is created using the *newAgent* operation, his components are instantiated from an already defined class (rule 4.5).

$$\frac{\langle \alpha, newAgent(\beta, \oslash, K, G, \oslash, \oslash, C, P, \oslash, \oslash), \oslash \rangle}{\langle \alpha, \oslash, \{\beta\} \rangle \quad \| \quad \langle \beta, \alpha, K, G, \oslash, \oslash, C, P, \oslash, \oslash \rangle} \tag{4.5}$$

An agent can completely remove one of his sub-agents:

$$\frac{\langle \alpha, \pi, kill(\beta), S_\alpha \rangle \parallel \langle \beta, \alpha \rangle, \text{where } \beta \in S_\alpha}{\langle \alpha, \pi, \oslash, S_\alpha - \{\beta\} \rangle} \tag{4.6}$$

Mobility operations

The mobility primitives are inspired from the ambient calculus. The reduction rules will be accompanied for these operations by a graphical representation that emphasizes the changes in the MAS hierarchy. Using *in*, an agent can enter another agent from the same level in the hierarchy (rule 4.7 and Figure 4.1) and using *out*, an agent can exit his parent (rule 4.8 and Figure 4.2). Unlike the ambient calculus, where there is no control, we added an asking/granting permission mechanism, represented in term of co-actions, in the same spirit with the *safe ambients* [136], with the main difference that one can specify the agent to whom he will grant a permission. By default, a CLAIM agent will receive these permissions, unless another agent is explicitly programmed to refuse to give them.

$$\frac{\langle \pi, S_\pi \rangle \parallel \langle \alpha, \pi, in(\beta) \rangle \parallel \langle \beta, \pi, \overline{in}(\alpha), S_\beta \rangle, \alpha, \beta \in S_\pi}{\langle \pi, S_\pi - \{\alpha\} \rangle \parallel \langle \alpha, \beta, \oslash \rangle \parallel \langle \beta, \pi, \oslash, S_\beta \cup \{\alpha\} \rangle} \tag{4.7}$$

$$\frac{\langle \pi, S_\pi \rangle \parallel \langle \alpha, \beta, out(\beta) \rangle \parallel \langle \beta, \pi, \overline{out}(\alpha), S_\beta \rangle, \beta \in S_\pi, \alpha \in S_\beta}{\langle \pi, S_\pi \cup \{\alpha\} \rangle \parallel \langle \alpha, \pi, \oslash \rangle \parallel \langle \beta, \pi, \oslash, S_\beta - \{\alpha\} \rangle} \tag{4.8}$$

In both cases, if the structural condition is not verified or if the agent does not receive the permission (*i.e.* the other does not have the correspondent co-action), the mobility process waits until the operation is possible.

The *move* mobility operation is a direct migration to another agent, without verifying a structure condition (rule 4.9 and Figure 4.3). Nevertheless, the operation is subject to the $\overline{in}$ and $\overline{out}$ permissions.

$$\frac{\langle \pi, \overline{out}(\alpha), S_\pi \rangle \parallel \langle \alpha, \pi, move(\beta), S_\alpha \rangle \parallel \langle \beta, \overline{in}(\alpha), S_\beta \rangle, \alpha \in S_\pi}{\langle \pi, \oslash, S_\pi - \{\alpha\} \rangle \parallel \langle \alpha, \beta, \oslash, S_\alpha \rangle \parallel \langle \beta, \oslash, S_\beta \cup \{\alpha\} \rangle} \tag{4.9}$$

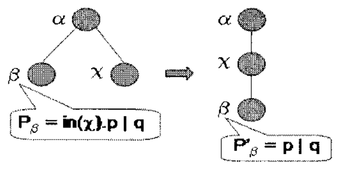


Figure 4.1. The enter operation

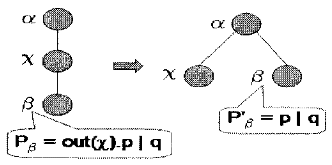


Figure 4.2. The exit operation

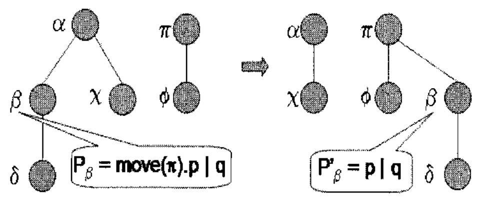


Figure 4.3. The move operation

The *open* and *acid* actions are used as in the original ambient calculus, respectively for opening one of the sub-agents (rule 4.10 and Figure 4.4), and for opening his own boundaries (rule 4.11 and Figure 4.5). Nevertheless, they have been adapted to intelligent agents. Hence, not only the running processes and the sub-agents of the open agent, but also his knowledge base and capabilities, become components of his parent. In this way, an agent can dynamically gather new knowledge and capabilities and can adapt himself to the requirements of an application. These operations are controlled by co-actions and allow a dynamic reconfiguration of a MAS.

$$\frac{\begin{array}{c}\langle \alpha, K_\alpha, C_\alpha, P \mid open(\beta), S_\alpha\rangle \parallel \langle \beta, \alpha, K_\beta, C_\beta, Q \mid \overline{open}(\alpha), S_\beta\rangle \parallel a_\beta \\ \text{where } \beta \in S_\alpha, a_\beta = \langle \gamma_\beta, \beta\rangle, \forall \gamma_\beta \in S_\beta\end{array}}{\langle \alpha, K_\alpha \cup K_\beta, C_\alpha \cup C_\beta, P \mid Q, S_\alpha \cup S_\beta\rangle \parallel a_\beta, \text{where } a_\beta = \langle \gamma_\beta, \alpha\rangle, \forall \gamma_\beta \in S_\beta} \tag{4.10}$$

$$\frac{\begin{array}{c}\langle \alpha, K_\alpha, C_\alpha, P \mid \overline{acid}(\beta), S_\alpha\rangle \parallel \langle \beta, \alpha, K_\beta, C_\beta, Q \mid acid, S_\beta\rangle \parallel a_\beta \\ \text{where } \beta \in S_\alpha, a_\beta = \langle \gamma_\beta, \beta\rangle, \forall \gamma_\beta \in S_\beta\end{array}}{\langle \alpha, K_\alpha \cup K_\beta, C_\alpha \cup C_\beta, P \mid Q, S_\alpha \cup S_\beta\rangle \parallel a_\beta, \text{where } a_\beta = \langle \gamma_\beta, \alpha\rangle, \forall \gamma_\beta \in S_\beta} \tag{4.11}$$

All these mobility operations are considered atomics at the semantical level and are executed in one step.

Instructions

There are two instructions in CLAIM. The first one, *forAllKnowledge*, allows to sequentially execute a process for all the elements in the knowledge base verifying a criterion (rule 4.12). The second instruction, *forAllAgents*, allows to execute a process for all the sub-agents verifying a certain criterion (*e.g.* all sub-agents - rule 4.13, or all sub-agents belonging to a specific class

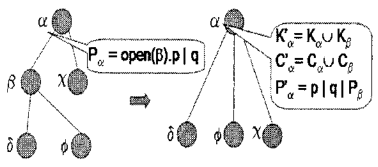

α
β
χ
δ
φ
P_α = open(β).p | q
α
$K'_\alpha = K_\alpha \cup K_\beta$
$C'_\alpha = C_\alpha \cup C_\beta$
P'_α = p | q | P_β
δ
φ
χ

Figure 4.4. The open operation

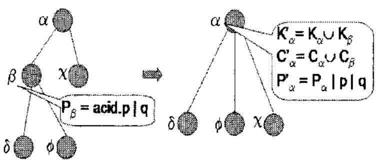

α
β
χ
δ
φ
P_β = acid.p | q
α
$K'_\alpha = K_\alpha \cup K_\beta$
$C'_\alpha = C_\alpha \cup C_\beta$
$P'_\alpha = P_\alpha$ | p | q
δ
φ
χ

Figure 4.5. The acid operation

- rule 4.14). The notation $p_i\{x_i/x\}$ symbolizes the substitution of all the occurrences of x with x_i (or of the variables in x with corresponding values from x_i) in p.

$$\frac{\langle \alpha, K, forAllKnowledge(k)\{p_i\}\rangle}{\langle \alpha, K, p_i\{k_1/k\}....p_i\{k_j/k\}\rangle, \forall k_i \in K, 1 \leq i \leq j, k \cong k_i} \tag{4.12}$$

$$\frac{\langle \alpha, K, forAllAgents(?x)\{p_i\}, S\rangle}{\langle \alpha, K, p_i\{\gamma_1/\gamma\}....p_i\{\gamma_n/\gamma\}, S\rangle, \forall \gamma_i \in S, 1 \leq i \leq n} \tag{4.13}$$

$$\frac{\langle \alpha, K, forAllAgents(?x : cl)\{p_i\}, S\rangle, S_{cl} \subseteq S, \forall \gamma_i \in S_{cl}, \gamma_i : cl, \gamma_i \text{ belongs to the class } cl}{\langle \alpha, K, p_i\{\gamma_1/\gamma\}....p_i\{\gamma_j/\gamma\}, S\rangle, \forall \gamma_i \in S_{cl}, 1 \leq i \leq j} \tag{4.14}$$

Updating effects

The effects are added in the effect list after the successful execution of the capability's process. If the achieved effects correspond to goals, they will be removed from the lists of not treated and processing goals.

$$\frac{\langle \alpha, G, G', addEffect(e_i), E\rangle}{\langle \alpha, G - \{e_i\}, G' - \{e_i\}, \oslash, E \cup \{e_i\}\rangle} \tag{4.15}$$

The goal-driven behavior

Concurrently with the reactive behavior, in which processes are executed when messages are received, an agent has a proactive behavior, accomplished using the capabilities' effects. When a capability has an effect corresponding to one of his goals, the agent will try to execute the capability. If its condition is true, the corresponding process is executed, (rule 4.16, where p'_i, Ω'_i and $e'_1...e'_j$ have the variables replaced with values from g).

$$\frac{\langle \alpha, \{g\}, \oslash, C, \oslash\rangle, \exists\langle s_i, m_i, \Omega_i, p_i, E_i\rangle \in C, \exists e_i \in E_i, e_i \cong g, V(\Omega'_i) = true}{\langle \alpha, \oslash, \{g\}, C, p'_i.addEffect(e'_1)....addEffect(e'_j)\rangle, e'_1...e'_j \in E_i} \tag{4.16}$$

If the condition allowing to achieve the goal contains an agent' effect not achieved yet, the agent will try first to achieve this effect, by adding it in his goals list. In the same time, the fist goal is moved from the current goals list to the processing goals (rule 4.17).

$$\frac{\langle \alpha, \{g\}, \oslash, C\rangle, \exists \langle s_i, m_i, \Omega_i, p_i, E_i\rangle \in C, \exists e_i \in E_i, e_i \cong g, \quad V(\Omega_i) = false, \Omega_i \text{ contains } this.e_i}{\langle \alpha, \{e_i'\}, \{g\}, C\rangle} \tag{4.17}$$

If the condition allowing to achieve the goal contains an effect of another agent, the effect is requested to the other agent using a specific message, *askEffect* (rule 4.18).

$$\frac{\langle \alpha, \{g\}, \oslash, C, \oslash\rangle, \exists \langle s_i, m_i, \Omega_i, p_i, E_i\rangle \in C, \exists e_i \in E_i, e_i \cong g, \quad V(\Omega_i) = false, \Omega_i \text{ contains } \beta.e_i}{\langle \alpha, \oslash, \{g\}, C, send(\beta, askEffect(e_i'))\rangle} \tag{4.18}$$

When an agent receives an *askEffect* message, if he does not have a capability with this message, meaning that the agent is programmed to treat differently the requests for services from other agents, he will add the demanded effect to his list of goals (rule 4.19).

$$\langle \beta, \oslash, \alpha\{askEffect(e_i)\}\rangle \quad \rightarrow \quad \langle \beta, \{\alpha.e_i\}, \oslash\rangle \tag{4.19}$$

The treatment of this new goal, resulting from another agent's demand, is done in the same way as his own goals. The only difference is that after the successful achievement of this external goal, a *doneEffect* message is sent to the agent that requested it (rule 4.20).

$$\frac{\langle \beta, G, G', addEffect(e_i), E\rangle, \text{ and } \exists \alpha.e_i \in G \text{ or } \exists \alpha.e_i \in G'}{\langle \beta, G - \{e_i\}, G' - \{e_i\}, send(\alpha, doneEffect(e_i)), E \cup \{e_i\}\rangle} \tag{4.20}$$

The treatment of a *doneEffect* message consists in removing the effect from the goals lists and adding it in the effect list, similar with the *addEffect* process.

Variable instantiation

The language allows to instantiate variables that will be used in the following processes in the current sequence (rule 4.21).

$$\langle \alpha, ?x = v.p_i\rangle \quad \rightarrow \quad \langle \alpha, p_i\{v/?x\}\rangle \tag{4.21}$$

Sequence

If an agent can evolve from a state containing a process p_i into another state containing the process p_i', then the agent containing p_i followed (in sequence) by another process q is able to evolve into p_i' followed by q.

$$\text{if } \langle \alpha, p_i\rangle \rightarrow \langle \alpha, p_i'\rangle \text{ then } \langle \alpha, p_i.q\rangle \rightarrow \langle \alpha, p_i'.q\rangle \tag{4.22}$$

Java and Web Services

As seen in the previous section, the programming language offers additional features, for calling Java methods or for invoking Web Services, that cannot change the components of an agent and we do not treat them at the semantical level.

Verification of programs: a discussion

The operational semantics presented above is just a first necessary step towards the formal verification of multi-agent programs written in CLAIM. The formal definition of an agent is more complex than the other formalisms treating mobile processes and the verification become much more complicated. We are currently studying aspects as programs' correctness (desirable properties that programs should verify [5]) and verification and we provide here a brief discussion about the characteristics of CLAIM programs. A CLAIM program is distributed and concurrent, containing agent communicating asynchronously and that do not share common variables. We have already presented the notion of program termination. We continue in this section with other important properties.
Determinism: A program is determinist if for any given state, there is exactly one next possible computational state. CLAIM programs are implicitly non-deterministic, because starting from a state, a program can evolve in several different states (see below).

The next configuration is a valid CLAIM program.

$$\begin{array}{c}\langle \tau, S_\tau \cup \{\pi\}\rangle \;\; \| \;\; \langle \pi, \overline{out}(\beta).p_k, S_\pi \cup \{\alpha, \beta\}\rangle \;\; \| \\ \langle \alpha, \pi, in(\beta).p_i\rangle \;\; \| \;\; \langle \beta, \pi, out(\pi).p_l \mid \overline{in}(\alpha).p_j, S_\beta\rangle\end{array}$$

This configuration can evolve (with equal probabilities) in two different configurations. If α executes *in*:

$$\begin{array}{c}\langle \tau, S_\tau \cup \{\pi\}\rangle \;\; \| \;\; \langle \pi, \overline{out}(\beta).p_k, S_\pi \cup \{\beta\}\rangle \;\; \| \\ \langle \alpha, \beta, p_i\rangle \;\; \| \;\; \langle \beta, \pi, out(\pi).p_l \mid p_j, S_\beta \cup \{\alpha\}\rangle\end{array}$$

or, if *out* is executed by β:

$$\begin{array}{c}\langle \tau, S_\tau \cup \{\pi, \beta\}\rangle \;\; \| \;\; \langle \pi, p_k, S_\pi \cup \{\alpha\}\rangle \;\; \| \\ \langle \alpha, \pi, in(\beta).p_i\rangle \;\; \| \;\; \langle \beta, \tau, p_l \mid \overline{in}(\alpha).p_j, S_\beta\rangle\end{array}$$

In the first case, β will still be capable of executing $out(\pi)$, but in the second case, α no longer can enter β, because he is not at the same level in the hierarchy anymore. Nevertheless, we guarantee at the implementation level that this kind of program will evolve in a stable state (one of the two in our example), in concordance with the reduction rules.
Deadlock: A configuration of a program is called deadlock if the configuration is non-terminal and there is no possible successor configuration (using a reduction rule). In CLAIM, because of the needed structure condition

for the mobility operation, an agent may try infinitely to execute an *in* operation, for entering an agent that is not in his neighborhood (and may never be), and consequently the next processes (in the same sequence) are blocked. However, we are not considering this as being a deadlock configuration, because the destination agent may be sometimes in the future in the neighborhood thus verifying the structural condition and unblocking the execution.

Correctness: A program is correct if it satisfies the intended input-output relation. To prove the correctness of CLAIM programs in syntax-directed manner, we are using a proof system. A proof system is a finite set of axiom schemas and proof rules. An axiom is a correctness formula representing the intended next states of a program starting from initial states. These axioms correspond to the reduction rules introduced earlier (note that we did not present in this chapter all the reduction rules; however, the proof system contains them all). A correctness formula is true with respect to the operational semantics reduction rules. Our current work tackles the soundness and the completeness of the proof system.

Structural congruence: As a first step towards the verification of MAS built using CLAIM, we studied the structural congruence of programs. We defined a CLAIM program as a set of running agents. Two *programs* are equivalent if they exhibit an identical behavior for an external observer. Following this reasoning, two programs are equivalent if they have *equivalent running agents.* That is, the same agents, with the same name, parent, knowledge base, goals, messages, capabilities and with *equivalent running processes.* So, the equivalence between programs is reduced at equivalence between processes inside agents. Processes are grouped into equivalence classes using the structural congruence relation $\equiv$. Its properties are presented below.

$$\begin{array}{l|l}
p \equiv p & p \mid q \equiv q \mid p \\
p \equiv q \Rightarrow q \equiv p & (p \mid q) \mid r \equiv p \mid (q \mid r) \\
p \equiv q, q \equiv r \Rightarrow p \equiv r & p \equiv q \Rightarrow p \mid r \equiv q \mid r \\
p \mid 0 \equiv p & p \equiv q \Rightarrow p.r \equiv q.r \\
p.0 \equiv p & p \equiv q \Rightarrow r.p \equiv r.q
\end{array}$$

4.2.3 Software Engineering Issues

The language includes the notion of class of agents. Generic classes can be defined and instantiated later. In this version of the language there is no inheritance as in object-oriented programming, but we intend to offer the possibility to define classes of agents that are sub-classes (specializations) of other classes. Nevertheless, at the agent level, CLAIM offers two primitives, *open* and *acid*, allowing an agent to gather sub-agents, processes, knowledge and capabilities from an open sub-agent, thus allowing a dynamic reconfiguring and adaptability of a MAS. We also developed several libraries of classes of agents for different domains, that can be parameterized and used by designers.

The CLAIM agents can invoke Java methods or Web Services for computational purposes. In the future, we intend to give the agents the possibility to invoke methods or programs implemented in other programming languages.

4.2.4 Other features of the language

The lack of formalisms to deal with both intelligent and mobile agents was one of our main motivations in developing CLAIM. The agents' mobility is a central aspect in our framework. We can easily model agents' reasoning, but our target applications must take advantage of both mobility and cognitive skills. There is a strong mobility at the agents' processes level and a week mobility for the invoked Java methods.

Concerning the extensibility of the language, the main constructs of CLAIM (*e.g.* agents' creation, mobility and communication primitives) are fixed. Nevertheless, the language offers the possibility to the agents' designer to develop his own ontology for representing knowledge or goals and for creating his own messages, with a specific treatment (represented by capabilities), to suit the current application.

4.3 Platform

The CLAIM language is supported by a dedicated platform, called SyMPA (French: Système Multi-Plateforme d'Agents), implemented in Java and that offers all the necessary mechanisms needed for the design and the secure execution of a distributed MAS.

4.3.1 Available tools and documentation

There are many platforms for mobile agents nowadays. The main difference of SyMPA with respect to other mobile agents platforms is that it supports agents implemented in CLAIM, an agent-oriented programming language while the other platforms support agents implemented using mainly object-oriented languages (*e.g.* Java in most cases). In addition, a CLAIM agent deployed in SyMPA can use Java methods. SyMPA is compliant with the specifications of the MASIF [151] standard from OMG, that provides a set of interfaces and definitions for the mobile agents' management, identification, authentication, localization, tracking, communication, mobility and security.

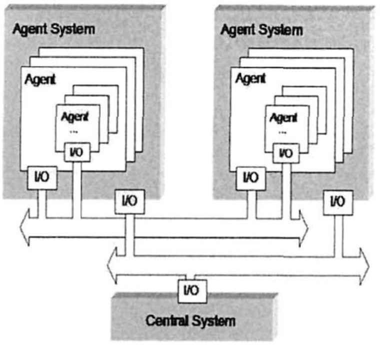


Figure 4.6. SyMPA's Architecture

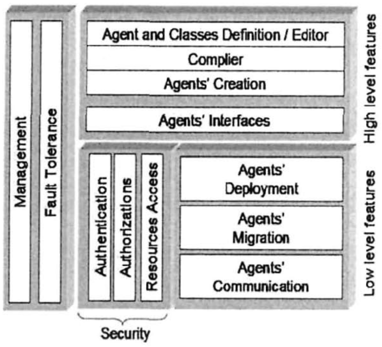


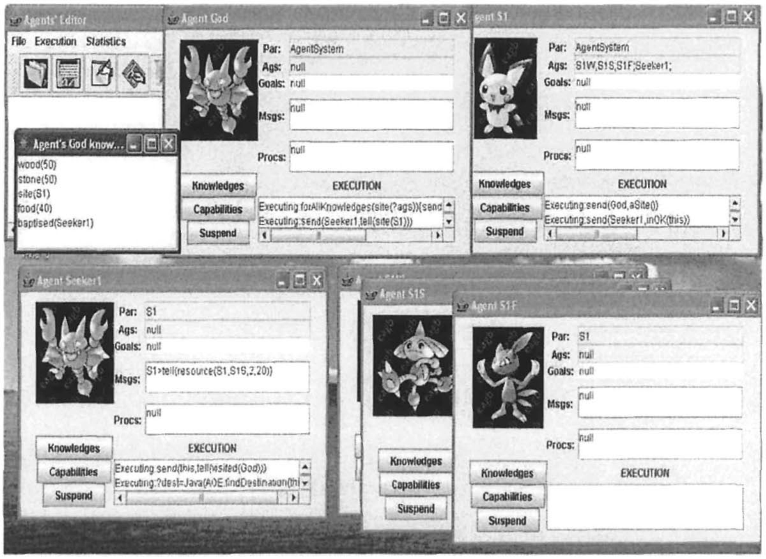
Figure 4.7. SyMPA's features

SyMPA can be deployed on a set of connected computers. It provides installation and deployment guidelines and a tutorial is currently developed. The platform's architecture is presented in Figure 4.6. There is a central system providing management functions. An agent system is deployed on each computer connected to the platform. It provides a graphical interface for defining and creating agents and for visualizing their execution, a compiler, mechanisms for agents' deployment, communication, migration and management (conf. Figure 4.7), all of these in a secure and fault tolerant environment. The compiler was implemented using JavaCC (Java Compiler Compiler) [84].

The agent system is also in charge of the communication with other agent systems or with the central system and of the mobility. The communication and the mobility are implemented using Java on top of the TCP/IP protocol. For each running agent, a optional graphical interface (Figure 4.8) can be used to monitor his behavior, communication or mobility.

Mobility

Due to the hierarchical representation of the agents and the distributed deployment of an MAS, we distinguish local and remote migrations. The local migration takes place inside a hierarchy, while the remote migration is the migration between hierarchies, using the *move* primitive.
The remote mobility in SyMPA can be considered at two levels. First, there is a strong migration at the language level, because, before the migration, the state of an agent is saved and then transferred to the destination. The agent's



```
wood(50)
stone(50)
site(S1)
food(40)
baptised(Seeker1)
```

Agent God

```
Par:   AgentSystem
Ags:   null
Goals: null
Msgs:  null
Procs: null
Executing forAllKnowledges(site(?ags)){send
Executing:send(Seeker1,tell(site(S1)))
```

Agent S1

```
Par:   AgentSystem
Ags:   S1W,S1S,S1F;Seeker1;
Goals: null
Msgs:  null
Procs: null
Executing:send(God,aSite())
Executing:send(Seeker1,inOK(this))
```

Agent Seeker1

```
Par:   S1
Ags:   null
Goals: null
Msgs:  S1>tell(resource(S1,S1S,2,20))
Procs: null
Executing send(this,tell(visited(God)))
Executing:?dest=Java(AOE.findDestination(thi
```

Agent S1F

```
Par:   S1
Ags:   null
Goals: null
Msgs:  null
Procs: null
```

Figure 4.8. Agents' interfaces

language-specific processes are resumed from their interruption point. An agent can be at any moment saved in a format similar to the definition, containing the current state (*e.g.* knowledge, messages, running processes). This representation is sent through the network to the destination agent system, in an encrypted format and the agent's execution is resumed from the saved state.

At the Java level, we use its mobility facilities, so there is a weak migration. A Java method that has begun before the migration will be reinvoked after the arrival at the destination. Since the migration is achieved using the language's primitives, unlike in other platforms, where there are Java objects that migrate during their execution, a solution can also be to let all the agent's running Java methods terminate before his migration.

Security

The mobile agents are programs running in a distributed and insecure environment (*e.g.* the Internet) where there are possible different attacks from the agents against the host agent system or attacks against an agent during the migration or during his execution. Several solutions exist against these attacks [101]. For the agent systems' protection, we are using agents' authentication, the control of the access to the system's resources in accordance with a set of permissions given to agents with regard to their authority, and audit techniques. For the agents' protection, we are using encryption during the migration and during the execution on a agent system (when the agent is stored on the disk), and also fault-tolerance mechanisms. The reader can find in [211] a detailed description of these security aspects.

4.3.2 Standards compliance, interoperability and portability

The SyMPA platform is implemented in Java and takes advantage of the portability and the platform-independence of this language. The SyMPA environment is composed of an ensemble of packages that can be installed on every computer with an operating system supporting *Java Virtual Machine*. After installing the packages, a few configuring operations are needed and the CLAIM language supported by the platform is ready to be used to implement MAS applications. We easily installed and tested the platform on Windows, Unix-based or Macintosh systems.

As we have already specified, SyMPA is compliant with the specifications of the MASIF [151] standard from the OMG, that provides a set of interfaces and definitions for the mobile agents' management, identification, authentication, localization, tracking, communication, mobility and security.

We have seen that CLAIM offers a set of agent-specific concepts and primitives for the agents' reasoning, communication and mobility. In addition, an agent can use Java methods or Web Services invocations for computational purposes.

Considering that the interoperability between heterogenous agents is a very important aspect in the MAS applications, we used the Web Services approach to develop an interoperability environment, called Web-MASI [80]. This environment is based on two key elements: an architecture that includes the MAS in the functional model of the Web Services and an interoperability module playing the role of interface between the agents and the Web Services layer. Using this plug-in module, the agents can publish their capabilities as Web Services, that can be invoked by other agents, independently from conceptual (agent architecture, interaction model) or technical (platform, programming language) characteristics.

4.3.3 Other features of the platform

The implementation of the platform is in a prototype stage, in continuous development and optimization and has already been used to implement several applications, presented in the next section. The results are very promising and an open-source version will be available soon, that will allow us to improve our implementation and to detect the expressiveness and the power but also the limits of the language and of the platform.

The developed applications cover a wide area, starting from simple applications with a small number of agents to largely distributed applications, with big number of highly communicating mobile agents. Concerning the reached performances, we could deploy up to 30 agents on one computer, but this number could easily increase if the resources consuming graphical interfaces of agents are not used. Nevertheless, in our current applications we used the interfaces to monitor the agents' execution, behavior, communication and migration. Concerning the scale of tests, until now we developed application using agents deployed on up to 10 connected computers.

As specified before, there is a central system with management functions in our environment. In the first phases, the central system had some problems with treating a great number of messages, but after adding fault-tolerance techniques and optimizations, the communications proceeded in a satisfying manner. Nevertheless, we are studying the possibility to introduce different management solutions (*e.g.* distributed, non-centralized) that the developer can choose in function of the current application's requirements.

The code reutilization is another of our priorities. The notion of class in central in our framework. Our long term goal is to have different already defined classes of agents for different types of applications that can be only parameterized and easily used by the designers.

4.4 Applications supported by the language and/or the platform

The CLAIM language supported by the SyMPA platform has been used to develop several applications, summarized below, that emphasize the main features of the framework, show the expressiveness and the facility of usage of the language and the robustness of the platform.

Translations

In the first phase of development of the CLAIM language, applications from other agent-oriented programming languages, such as *Airline reservations* from *AGENT-0* [206] or *Bolts Make Scenario* from *AgentSpeak* [233], were translated. FIPA protocols were also programmed using CLAIM. There is no mobility in these applications, but the agents' reasoning and communication were easily translated.

Research of information

One of the first applications implemented was the research of information on a network [82] using mobile agents. Receiving requests from users, these agents migrate to all the available connected sites searching for pieces of information corresponding to a request.

Electronic commerce

A more complex application, that justified the hierarchical representation of agents, was an e-commerce application [81], where there are several electronic markets distributed on a network. Each e-market has various departments (represented as sub-agents of a market), for different types of products. The markets can move with all the sub-departments to other sites in order to find clients and the clients can move to different markets searching for products.

Load balancing

In the two applications previously presented, the intelligent elements of the agents were central. An application focused on the computational aspects was implemented next. Thus, CLAIM and SyMPA served for programming an application of load balancing and resource sharing [212]. The connected computers' characteristics are gathered by mobile agents and the computers are classified using different criterions. The users' tasks are executed on computers satisfying some requirements and can dynamically migrate during the execution in order to finish the execution in the fastest way possible.

E-libraries network

The next step was to combine the intelligent features of the agents with the results of the load balancing application in an application containing a network of distributed cooperative digital libraries [129]. The libraries have sections and are used by customers searching for various documents. The libraries manage the subscribers, the documents and have information about other libraries, as the goal is to satisfy the customers, even if this means to direct them towards other libraries. A library can also distribute one or several sections to another site when there are too many clients on the local computer, using results from the load-balancing application.

Veracruz coffee market

Another complex application developed using CLAIM was the modelling of the coffee market in Veracruz, Mexico [213]. Using our framework, all the involved actors were designed, proposing an agent-based application able to deal with the different types of transaction negotiations and covering the entire value chain of coffee.

A Case Study

In order to illustrate the language's specifications, we present here an application inspired from strategy games, such as *Age of Empires*[4]. As a simplified version, there is a village of people in a prehistoric era, trying to survive by gathering resources. There are sites of resources distributed on several computers of a network. Each site can contain three types of resources: wood, stone and food. The population is represented by a *Creator* agent that can create *Seeker* agents and resource gatherer agents for each type of resource (resources are consumed when creating new agents): *WoodCutter*, *Miner* and *Hunter*. Each type of agent has capabilities for gathering only his corresponding resource. The goal is to gather all the resources. We implemented several strategies, in order to observe the agents' behavior in different situations. Since the goal here is only to show examples of agents implemented in CLAIM, we focus on one scenario.

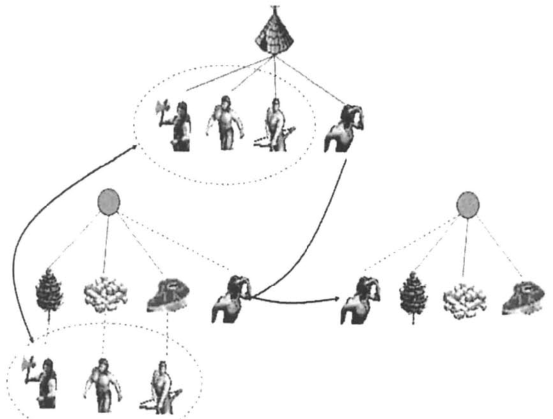


Figure 4.9. Application's schema

The *Creator* agent creates (using ***newAgent***) a *Seeker* agent, finds out the list of the existing sites and tells to the *Seeker* to migrate to each of them (using ***move***). When the *Seeker* arrives on a site, he "counts" the available

[4]http://www.microsoft.com/games/empires/

resources and asks (using ***send***) specialized agents from the *Creator*, who will create (using ***newAgent***) one specialized agent for each type of resource, agents that migrate to the specific resource agents on the site. After gathering the resources, they return to the village, give the resources to the *Creator* and wait for other calls. Meanwhile, the *Seeker* moves to other sites, searches for resources and asks for specialized agents. If there is no specialized agent available at the *Creator* when a new ask for help arrives, a new specialized agent is created.

We present only some of the most important capabilities of our agents. Every identified actor of our scenario will be represented as a class of agents. When programming a CLAIM class, one must identify the possible parameters of the class, the knowledge ontology (that can evolve during the execution), the chosen type of reasoning (forward reasoning or proactive or both), the goals (for agents with a proactive behavior), the capabilities, the messages to be exchanged with other agents and the necessary Java methods used for various computations.

The agents in the presented scenario use a forward reasoning, *i.e.* they execute actions when specific messages arrive and some (optional) conditions are verified. The *Creator* has an initial amount of resources, given as parameters for the class and represented in the knowledge base as *wood(?woodQuant)*, *stone(?stoneQuant)* and *food(?foodQuant)*. The quantities of resource evolve during the execution (decrease when new agents are created and increase when resources are brought by agents). Other manipulated knowledge represents the found sites (not known *a priori*). Several Java methods were needed for verifying if the agent has sufficient resources when he tries to create a new agent, for waiting an amount of time, for updating the quantities of resources, etc.

```
defineAgentClass Creator(?w,?s,?f) {
   authority=null;    parent=null;
   knowledge={wood(?w);stone(?s);food(?f);}
   goals=null;    messages=null;
   capabilities {
    findSites {
```

capability for sending to all the existing *Site* agents a message for asking their names; the *Site* agents which answer to this messages are added in the *Creator*' knowledge base

```
     message=findSites();
     condition=null;
     do{send(?agS:Site(),askSiteName()).Java(AOE.wait(30)).send(this,initSearch())}
     effects=null;
   }
    createSeeker {
```

capability for creating a *Seeker* (if there are sufficient resources), for telling him the names of the known sites and for requesting his departure

```
    message=initSearch();
    condition=Java(AOE.hasResources(this,0,0));
    do{?n=Java(AOE.baptise(this,0)).newAgent ?n:Seeker().
       forAllKnowledge(site(?ags)){send(?n,tell(site(?ags)))}.send(?n,seek())}
    effects=null;
   } ...
```

the class has several other capabilities for creating specialized agents when the *Seeker* arrives on a site and requests help and for updating the resources when these agents return.

```
  }
  processes={send(this,findSites())}
  agents=null;
}
```

The *Site* class has parameters representing the amount of each resource (the knowledge base contains pieces of information similar with those of a *Creator*) and capabilities for creating the sub-resource agents and for answering the questions concerning his names and his resources. The sub-resource agents are represented in a simple class (named *Resource*) that can receive agents and updates the amount of resources after a gatherer agent's passage.

A *Seeker* manipulates pieces of information about the known sites, about the visited sites and about the sites' resources. When created, he selects a destination site (known and not visited already; he uses a Java method for this), migrates to this site, finds out the amount of available resources (by communicating with the site agent) and then requests specialized agents from the *Creator*.

```
defineAgentClass Seeker() {
  authority=null;  parent=null;  knowledge=null;  goals=null;  messages=null;
  capabilities {
   seek {
```

capability for migrating to a not visited site and for asking the amount of available resources

```
    message=seek();
    condition=null;
    do{?d=Java(AOE.findDestination(this)).move(this,?d).send(?d,needResources(?d))}
    effects=null;
   } ...
```

he requests next specialized agents and continues the search migrating to other sites.

```
  }
  processes=null;   agent=null;
}
```

The specialized gatherer agents (*WoodCutter, Miner* and *Hunter*) can migrate to specific *Resource* agents, return to the *Creator*, give him the gathered resources and await for new requests.

After defining all the classes of agents for our scenario (but also for the other considered scenarios) and writing all the necessary Java methods, the SyMPA platform was deployed on several computers of the network. Several sites of resources were started on different computers and a *Creator*. We observed the behavior of all the agents in our application (not only for presented scenario) that migrate in order to gather resources and we also counted the times for gathering all the resources and the *Creator*' resource variation for different scenarios.

4.5 Final Remarks

In this chapter, we argue that the development of MAS applications needs specific languages (*i.e.* agent-oriented) in order to reduce the gap between the design and the implementation phases.

The presented language, CLAIM, frees the designer from time-consuming implementation aspects and combines in a unified framework the advantages of the intelligent agents with those of the ambient calculus (particularly suitable for mobile computation). Hence, both computational aspects (communication, mobility, processing) and cognitive features (knowledge, goals and reasoning) of agents are easily represented thanks to CLAIM.

For using the language in real-life applications, we would like to be able to verify some important aspects of the built MAS, using a formal operational semantics, whose main elements were also presented in this chapter.

Using a flexible hierarchical topology of the MAS, a goal-driven behavior and a mental state of agents that continuously evolves in an autonomous manner, CLAIM allows a dynamic re-configuring of the built MAS in order to give the system the full scope to adapt its structure and to meet the requirements of target applications.

The language is supported by a distributed platform, SyMPA, that offers all the necessary mechanisms for creating and deploying CLAIM agents and for a secure execution of a distributed MAS.

CLAIM and SyMPA have been used for developing several complex applications that showed the expressiveness of the language and the robustness and the strength of the platform, such as an application for information research on the Web, electronic commerce applications, a load balancing and resource sharing application using mobile agents or an application of a digital libraries network. All the results were very promising.

The current work tackles the verification of CLAIM programs, using the defined operational semantics, the optimization of the platforms and the adaptability and interoperability issues. We would like to deploy SyMPA on mobile devices in order to fulfill the ambient intelligence requirements.

II

JAVA-BASED AGENT PROGRAMMING LANGUAGES