

Multi-Agent Programming Languages, Platforms and Applications

Summer 2006

Outline I

Abstract

This chapter describes **Jason**, an interpreter written in Java for an extended version of AgentSpeak, a logic-based agent-oriented programming language that is suitable for the implementation of reactive planning systems according to the BDI architecture. We describe both the language and the various features and tools available in the platform.

Keywords

Logic-Based Agent Programming, Beliefs-Desires-Intentions, Operational Semantics, Speech Acts, Plan Exchange, Java-based Extensibility/Customisation.

The golden fleece

*Now was remaining as the last conclusion of this game,
By force of chaunted herbes to make the watchfull Dragon sleepe
Within whose eyes came never winke: who had in charge to keepe
The goodly tree upon the which the golden fleeces hung.*

...

*The dreadfull Dragon by and by (whose eyes before that day
Wist never erst what sleeping ment) did fall so fast asleepe
That Jason safely tooke the fleece of golde that he did keepe.*

P. Ovidius Naso, Metamorphoses (ed. Arthur Golding), Book VII.

Authors

Rafael H. Bordini, Jomi F. Hübner and Renata Vieira

Results of research on MAS

- A **variety of techniques** for the development of complex distributed systems.
- These techniques work in environments that are traditionally thought to be too unpredictable for computer programs to handle.
- **But real-world MAS** is comparable to the **Golden Fleece**.
 \rightsquigarrow **Jason**

AgentSpeak(L)

- Is a **Elegant** logic-based agent-oriented programming language.
- Is for reactive planning systems (PRS).
- Is benefited from the experience of Beliefs-Desires-Intentions (BDI).
- Is only an **abstract** agent programming language.

Jason

- Is an extended AgentSpeak to a **practical** programming language with MAS techniques.
- Provides **operational semantics**.
- Is **for real-world applications**.
- **Jason** is available **Open Source** under GNU LGPL at <http://jason.sourceforge.net>.

Features available in Jason

- Speech-act based **inter-agent communication**.
- Annotations on plan labels, which can be used by elaborate selection functions
- The possibility to run a multi-agent system distributed over a network.
- Fully customisable selection functions, trust functions, and overall agent architecture.
- Straightforward extensibility by means of user-defined “internal actions”.
- Clear notion of **multi-agent environments**, which can be implemented in Java.

Scenario: The Airport Chronicle

Scenario

Year 2070 ad., London Heathrow, airports are now completely staffed by together networked robots:

- CPH903 robots - **c**ute, **p**olite **h**andy: Welcome people, bear luggage, provide informations.
- MDS79 robots - **m**ulti-**d**evice **s**ecurity: Very expensive, for bomb detection.
- MDS79 and CPH903 can cooperate to ensure that reported unattended luggage has been cleared away.

Unattended luggage

- MDS79 checks whether there is a bomb. If there is any chance of there being a bomb the robot sends a message to the specialised bomb-disarming team.
- Once these robots are called in, the MDS79 and CPH903 robots that had been relocated can go back to their normal duties.
- The bomb-disarming robots decide whether to set off a security alert to evacuate the airport, or alternatively they attempt to disarm the bomb or move it to a safe area, if they can ensure such courses of action would pose no threat to the population.
- In case the MDS79 robot detects no signs of a bomb in the unattended luggage, the job is passed on to the accompanying CPH903 robot.

Definition 1.1 (AgentSpeak Agent)

AgentSpeak Agent is defined by

Beliefs: Belief base, plan library.

Goals: Achievement goals ('!') , test goals ('?').

Events: They are related either to changes in beliefs due to perception of the environment, or to changes in the agent's goals that originate from the execution of plans triggered by previous events.

A **triggering event** defines which events can initiate the execution of plan.

Plans: written by the programmer, triggered by the **addition** ('+') or **deletion** ('-') of beliefs or goals (the “mental attitudes” of AgentSpeak agents).

AgentSpeak Plan

Head: formed from a **triggering event** (specifying the events for which that plan is **relevant**).

Context: Conjunction of **belief literals**.

Applicable: The conjunction of literals must be a **logical consequence** of that agent's **current beliefs**.

Body: Sequence of **basic actions** or **(sub)goals**.

Basic actions: Atomic operations the agent can perform so as to change the environment (atomic formulæ, **action symbols** rather than predicate symbols).

```
skill(plasticBomb).  
skill(bioBomb).  
~skill(nuclearBomb).  
  
safetyArea(field1).  
  
@p1  
+bomb(Terminal, Gate, BombType) : skill(BombType)  
  <- !go(Terminal, Gate);  
    disarm(BombType).  
  
@p2  
+bomb(Terminal, Gate, BombType) : ~skill(BombType)  
  <- !moveSafeArea(Terminal, Gate, BombType).
```

@p3

```
+bomb(Terminal, Gate, BombType) : not skill(BombType) &  
                                   not ~skill(BombType)  
  <- .broadcast(tell, alter).
```

@p4

```
+!moveSafeArea(T,G,Bomb) : true  
  <- ?safeArea(Place);  
      !discoverFreeCPH(FreeCPH);  
      .send(FreeCPH, achieve,  
             carryToSafePlace(T,G,Place,Bomb)).
```

⋮

Specifications and Syntactical Aspects

AgentSpeak syntax accepted by **Jason**

<Atom> is an identifier beginning with lowercase letter or '.'.

<VAR> (i.e., a variable) is an identifier beginning with an uppercase letter.

<NUMBER> is any integer or floating-point number.

<STRING> is any string enclosed in double quote characters as usual.

Main differences to the original AgentSpeak(L)

- Whereever an **atomic formulæ** was allowed in the original, here a **literal** is used instead.
- Either an atomic formulæ: $p(t_1, \dots, t_n)$, $n \geq 0$, or $\sim p(t_1, \dots, t_n)$, where ‘ \sim ’ denotes strong negation.
- Default negation is used in the context of plans, and is denoted by ‘not’ preceding a literal.
- The **context** is therefore a conjunction of **default literals**.
- **Terms** now can be variables, lists (with Prolog syntax), as well as integer or floating point numbers, and strings (enclosed in double quotes as usual).
- Any atomic formulæ can be treated as a term, and (bound) variables can be treated as literals
- Infix relational operators, as in Prolog, are allowed in plan contexts.

Atomic formulæ can have “annotations”

- List of terms enclosed in square brackets immediately following the formula.
- Within the **belief base** to register the source information. A term `source(s)` is used in the annotations for that purpose.
- `s` can be an agent's name or two special atoms, `percept` and `self`.
- **Initial beliefs** are assumed to be **internal beliefs** (i.e., as if they had a `[source(self)]` annotation), unless the belief has any explicit annotation given by the user.
- **Plans** also have labels. A plan label can now be any atomic formula, including annotations but have a predicate symbol of arity 0, as in `aLabel` or `anotherLabel[chanceSuccess(0.7), expectedPayoff(0.9)]`.

Events

- If action fails or there is no applicable plan for a subgoal in the plan being executed to handle an internal event with a goal addition $+!g$, then the whole failed plan is removed from the top of the intention and an internal event for $-!g$ associated with that same intention is generated.
- If the programmer provided a plan that has a triggering event matching $-!g$ and is applicable, such plan will be pushed on top of the intention, so the programmer can specify in the body of such plan how that particular failure is to be handled.
- If no such plan is available, the whole intention is discarded and a warning is printed out to the console.

Example 1.2

- For an agent that persist on a goal $!g$ for as long as there are applicable plans for $+!g$, suffices it to include a plan $-!g : \text{true} \leftarrow !g.$ in the plan library.
- The body can be empty.
- A goal is only removed from the body of a plan when the intended means chosen for that goal finishes successfully
- It is also simple to specify a plan which, under specific condition, chooses to drop the intention altogether (by means of a standard internal action mentioned below).

Internal actions

- Can be used both in the context and body of plans.
- Any action symbol starting with ‘.’, or having a ‘.’ anywhere, denotes an internal action.
- These are **user-defined actions** and **run internally** by the agent.
- Called “internal” to make a clear distinction with actions that appear in the body of a plan and which denote the actions an agent can perform in order to change the shared environment (in the usual jargon of the area, by means of its “effectors”).
- In **Jason**, internal actions are coded in Java.

agent → beliefs plans

beliefs → (literal ".") *
*N.B.: a semantic error is generated
if the literal was not ground.*

plans → (plan) +

plan → ["@" atomic_formula]
triggering_event ":" context "<-" body "."

literal → atomic_formula
| "~" atomic_formula
| <VAR>

triggering_event → "+" literal

| "-" literal

| "+" " !" literal

| "-" " !" literal

| "+" " ?" literal

| "-" " ?" literal

default_literal → literal

| "not" literal

| "not" "(" literal ")"

| term ("<" | "<=" | ">" | ">=" | "==" | "\"\\==" | "=") term

| literal ("==" | "\"\\==" | "=") literal

```
context      → "true"  
              | default_literal ( "&" default_literal ) *  
  
body         → "true"  
              | body_formula ( ";" body_formula ) *  
  
body_formula → literal  
              | " !" literal  
              | " ?" literal  
              | "+" literal  
              | "-" literal  
  
atomic_formula → <ATOM> [ "(" list_of_terms ")" ]  
                  [ "[" list_of_terms "]" ]  
  
list_of_terms  → term ( "," term ) *
```

```
term      → atomic_formula
           | list
           | <VAR>
           | <NUMBER>
           | <STRING>

list      → "["
           [ term ( ( "," term ) *
                 | "|" ( list | <VAR> )
                 )
           ] "]"
```

Figure 1: BNF of the AgentSpeak Extension Interpreted by Jason.

Semantics and Verification

Important characteristics of Jason:

- It implements **operational semantics** of an extension of AgentSpeak
- Having formal semantics also allowed us to give **precise definitions** for **practical notions of beliefs, desires, and intentions** in relation to running AgentSpeak agents, which in turn underlies the work on formal verification of AgentSpeak programs.
- The formal semantics, using structural operational semantics.

Informal Semantics

Besides the belief base and the plan library, the AgentSpeak interpreter also manages:

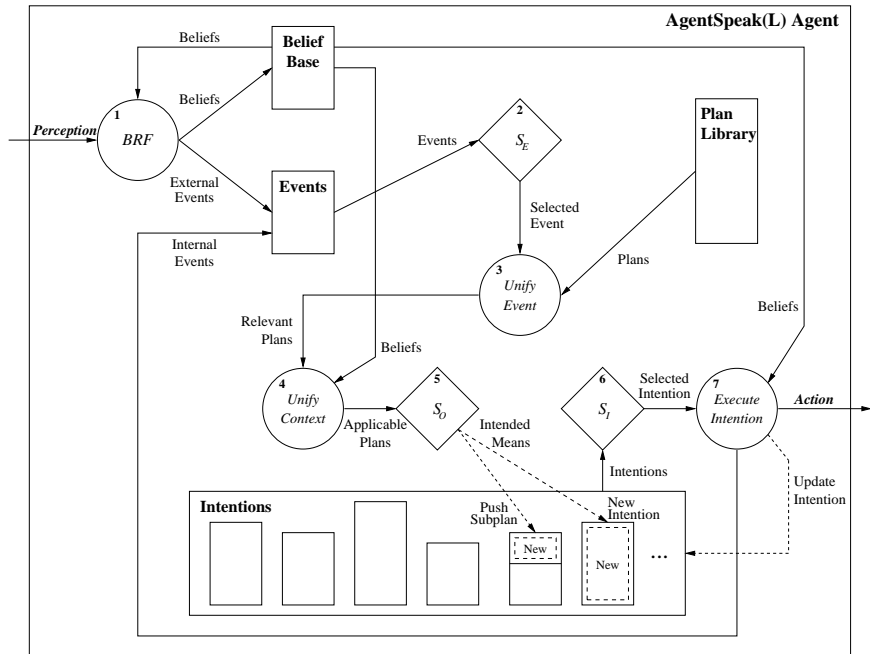
- A set of **events**;
- A set of **intentions**.

Its functioning requires three **selection functions**.

Event selection

- The **event selection function** (\mathcal{S}_E) selects a **single event** from the set of events.
- Another selection **function** (\mathcal{S}_O) selects an **“option”** (i.e., an applicable plan) from a set of applicable plans.
- a third selection **function** (\mathcal{S}_I) selects **one particular intention** from the set of intentions.
- The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent's characteristics.

- **Intentions** are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans.
- **Events**, which may start off the execution of plans that have relevant triggering events, can be:
 - **External** (originating from perception);
 - **Internal** (generated from the agent's own execution of a plan).



Interpretation Cycle of an AgentSpeak Program

- **Sets** (of beliefs, events, plans, and intentions) are represented as **rectangles**.
- **Diamonds** represent **selection** (of one element from a set).
- **Circles** represent some of the **processing** involved in the interpretation of AgentSpeak programs.
- At **every interpretation cycle** the **interpreter updates a list of events** (generated from perception or from the execution of intentions).
- It is assumed that beliefs are updated from perception and whenever there are changes in the agent's beliefs, this implies the insertion of an event in the set of events.
- This belief revision function is not part of the AgentSpeak interpreter, but rather a necessary component of the agent architecture.

- After \mathcal{S}_E has selected an event, the interpreter has to unify that event with triggering events in the heads of plans.
- This generates the set of all **relevant plans** for that event.
- By checking whether the context part of the plans in that set follows from the agent's beliefs, the set of **applicable plans** is determined.
- Then \mathcal{S}_O chooses a single applicable plan from that set, which becomes the **intended means** for handling that event, and either:
 - Pushes that plan on the top of an existing intention (if the event was an internal one);
 - Or creates a new intention in the set of intentions (if the event was external).

- All that remains to be done at this stage is to select a single intention to be executed in that cycle.
- The S_I function selects one of the agent's intentions.
- On the top of that intention there is a plan, and the formula in the beginning of its body is taken for execution.
- This implies that either:
 - A **basic action** is performed by the agent on its environment;
 - An **internal event** is generated (achievement goal);
 - Or a **test goal** is performed.

- If the **intention** is to **perform a basic action or a test goal**, the **set of intentions needs to be updated**.
- In the case of a **test goal**, **the belief base will be searched** for a **belief atom** that unifies with the atomic formula in the test goal.
- If that search succeeds, further variable instantiation will occur in the partially instantiated plan which contained that test goal.

- In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention.
- When all formulæ in the body of a plan have been removed, the whole plan is removed from the intention, and so is the achievement goal that generated it.
- This ends a cycle of execution, and everything is repeated all over again, initially checking the state of the environment after agents have acted upon it, then generating the relevant events, and so forth.

Formal Semantics

The operational semantics is given by a set of rules that define a transition relation between configurations $\langle ag, C, M, T, s \rangle$.

Agent program ag

An **agent program** ag is, as defined above, a set of beliefs and a set of plans.

Agent's circumstance C

An agent's circumstance C is a tuple $\langle I, E, A \rangle$ where:

- I is a set of intentions $\{i, i', \dots\}$. Each intention i is a stack of partially instantiated plans.
- E is a set of events $\{(te, i), (te', i'), \dots\}$. Each event is a pair (te, i) , where te is a triggering event and i is an intention (a stack of plans in case of an internal event or \top representing an external event).
- A is a set of actions to be performed in the environment.

M

M is a tuple $\langle In, Out, SI \rangle$ whose components register the following aspects of communicating agents:

- In is the mail inbox. Elements of this set have the form $\langle mid, id, ilf, cnt \rangle$, where:
 - mid is a message identifier;
 - id identifies the sender of the message;
 - ilf the illocutionary force of the message;
 - cnt its content.
- Out is the mail outbox. Same format as above, except that now id refers to the agent to which the message is to be sent.
- SI is used to keep track of intentions that were suspended due to the processing of communication messages.

T

T is the tuple $\langle R, Ap, \iota, \varepsilon, \rho \rangle$, used to keep **temporary information** that is required in subsequent stages within a single reasoning cycle; its components are:

- R for the set of **relevant plans** (for the event being handled).
- Ap for the set of **applicable plans** (the relevant plans whose context are true).
- ι , ε , and ρ keep record of a particular intention, event and applicable plan (respectively) being considered along the execution of an agent.

Current step s

The **current step s** within an agent's reasoning cycle is symbolically annotated by $s \in \{\text{ProcMsg}, \text{SelEv}, \text{RelPI}, \text{ApplPI}, \text{SelAppl}, \text{AddIM}, \text{SelInt}, \text{ExecInt}, \text{ClrInt}\}$, which stand for:

- 1 Processing a message from the agent's mail inbox;
- 2 Selecting an event from the set of events;
- 3 Retrieving all relevant plans;
- 4 Checking which of those are applicable;
- 5 Particular applicable plan (the intended means);
- 6 Selecting one;
- 7 Adding the new intended means to the set of intentions;
- 8 Selecting an intention;
- 9 Executing the select intention;
- 10 Clearing an intention or intended means that may have finished in the previous step.

In order to keep the semantic rules clear, we adopt the following notations:

- If C is an AgentSpeak agent circumstance, we write C_E to make reference to the component E of C . Similarly for all the other components of a configuration.
- We write $T_i = _$ (the underline symbol) to indicate that there is no intention being considered in the agent's execution. Similarly for T_p and T_ε .
- We write $i[p]$ to denote an intention i that has plan p on its top.
- In the general case, an agent's initial configuration is $\langle ag, C, M, T, \text{ProcMsg} \rangle$, where ag is as given by the agent program, and all components of C , M , and T are empty.

Updating the Set of Intentions

$$\frac{T_{\varepsilon} = \langle te, T \rangle \quad T_{\rho} = (p, \theta)}{\langle ag, C, M, T, \text{AddIM} \rangle \longrightarrow \langle ag, C', M, T, \text{SellInt} \rangle} \quad (\text{EXTEV})$$

$$\text{where: } C'_I = C_I \cup \{ [p\theta] \}$$

$$\frac{T_{\varepsilon} = \langle te, i \rangle \quad T_{\rho} = (p, \theta)}{\langle ag, C, M, T, \text{AddIM} \rangle \longrightarrow \langle ag, C', M, T, \text{SellInt} \rangle} \quad (\text{INTEV})$$

$$\text{where: } C'_I = C_I \cup \{ (i[p])\theta \}$$

- Rule EXTEV says that if the event ε is external (which is indicated by T in the intention associated to ε) a new intention is created and its single plan is the plan p annotated in the ρ component.
- If the event is internal, rule INTEV says that the plan in ρ should be put on top of the intention associated with the event.

Intention Selection

$$\frac{C_I \neq \{\} \quad S_I(C_I) = i}{\langle ag, C, M, T, \text{SelInt} \rangle \longrightarrow \langle ag, C, M, T', \text{ExecInt} \rangle} \quad (\text{INTSEL}_1)$$

$$\frac{\text{where: } T'_1 = i \quad C_I = \{\}}{\langle ag, C, M, T, \text{SelInt} \rangle \longrightarrow \langle ag, C, M, T, \text{ProcMsg} \rangle} \quad (\text{INTSEL}_2)$$

- Rule INTSEL_1 uses an agent-specific function (S_I) that selects an intention (i.e., a stack of plans) for processing.
- Rule INTSEL_2 takes care of the situation where the set of intentions is empty (in which case, the reasoning cycle is simply restarted).

Executing a Plan Body

Achievement Goals:

$$\begin{array}{c}
 T_1 = i[\text{head} \leftarrow !at; h] \\
 \hline
 \langle ag, C, M, T, \text{ExecInt} \rangle \longrightarrow \langle ag, C', M, T, \text{ProcMsg} \rangle \quad (\text{ACHIEVE}) \\
 \text{where: } \begin{array}{l} C'_E = C_E \cup \{ \langle +!at, i[\text{head} \leftarrow h] \rangle \} \\ C'_I = C_I \setminus \{ T_1 \} \end{array}
 \end{array}$$

- The plan being executed is always the one on top of the intention that has been previously selected.
- All the rules in this group discard the intention τ ; another intention can then be eventually selected.
- This rule registers a new internal event in the set of events E .
- The intention that generated the internal event is removed from the set of intentions C_I .

Executing a Plan Body

Updating Beliefs:

$$\frac{T_i = i[\text{head} \leftarrow +b; h]}{\langle ag, C, M, T, \text{ExecInt} \rangle \longrightarrow \langle ag', C', M, T, s \rangle} \quad (\text{ADDBEL})$$

$$\begin{aligned} \text{where: } ag'_{bs} &= ag_{bs} + b[\text{source}(\text{self})] \\ C'_E &= C_E \cup \{ \langle +b[\text{source}(\text{self})], T \rangle \} \\ C'_I &= (C_I \setminus \{ T_i \}) \cup \{ i[\text{head} \leftarrow h] \} \\ s &= \begin{cases} \text{ClrInt} & \text{if } h = \top \\ \text{ProcMsg} & \text{otherwise} \end{cases} \end{aligned}$$

- Rule ADDBEL adds a new event to the set of events E .
- Analogous rule DELBEL.

Verification

Jason is amenable to formal verification.

For AgentSpeak developed:

- **Model checking** techniques.
- **Statespace reduction** techniques.

Software Engineering Issues

- Accordance to the **Prometheus methodology**.
- **More readable** than JACK and Jadex.
- Internal actions - **high level representation** of the agent's reasoning
 \rightsquigarrow Far **more elegant** than with other agent programming languages
 (JACK, Jadex)

Other Features of the Language

Communication in AgentSpeak

- Inspired by KQML performatives.
- *s* Agent that sends the message.
- *r* Agent that receives the message.
- `tell`, `untell` can be used pro-actively or reactively.

- `tell`: s intends r to believe (that s believes) the sentence in the message's content to be true;
- `untell`: s intends r not to believe (that s believes) the sentence in the message's content to be true;
- `achieve`: s requests that r try to achieve a state of the world where the message content is true;
- `unachieve`: s requests that r try to drop the intention of achieving a state of the world where the message content is true;

`tellHow`: s informs r of a plan;

`untellHow`: s requests that r disregard a certain plan (i.e., delete that plan from its plan library);

`askIf`: s wants to know if the content of the message is true for r ;

`askAll`: s wants all of r 's answers to a question;

`askHow`: s wants all of r 's plans for a triggering event.

- A Mechanism for **receiving** and **sending messages asynchronously** is used.
- Messages are stored in mail box and **one** of them is **processed** by the agent at the beginning of a **reasoning cycle**.
- The particular message is determined by a **selection function**.
- **"Given" function** defines a set of **socially acceptable** messages.
(e.g. avoid mail-attacks, define trustfulness)

Example 1.3 (Customisation the power relation in Jason)

A CPH903 robot only does what an MDS79 robot asks.

```
package cph;
import jason.asSemantics.Agent;

public class CPHAgent extends Agent {

    public boolean socAcc(Message m) {
        if (m.getSender().startsWith("mds") &&
            m.getIlForce().equals("achieve")) {
            return true;
        } else {
            return false;
        }
    }
}
```

Annotations

- We annotate for **each belief** what is **its source**.
- The **annotation mechanism** provides a very **elegant notation**.
- It has advantages in terms of **expressive power** and **readability**.
- It allows the use of such explicit information in an agent's reasoning.
- **Belief sources** can be annotated so as to **identify** which was **the agent** that sent the informations,
- as well as **denote internal beliefs or percepts**.

Other Features of the Language

Cooperation in AgentSpeak

Coo-BDI (Cooperative BDI) extends traditional BDI agent-oriented programming languages in many respects:

- The introduction of **cooperation** among agents for the retrieval of external plans for a given triggering event.
- The extension of plans with **access specifiers**.
- The extension of **intentions** to take into account the external plan retrieval mechanism.
- The modification of the interpreter to cope with all these issues.

The **cooperation strategy** of an agent **Ag** includes

- the set of agents with which it is expected to cooperate,
- the plan retrieval policy,
- the plan acquisition policy.
- The cooperation strategy may evolve during time, allowing greater flexibility and autonomy to the agents.
- `trusted(Te, TrustedAgentSet)`, where *Te* is a (not necessarily ground) triggering event and *TrustedAgentSet* is the set of agents that *Ag* will contact in order to obtain plans relevant for *Te*.

- `retrievalPolicy(Te,Retrieval)`, where *Retrieval* may assume the values `always` and `noLocal`, meaning that relevant plans for the trigger *Te* must be retrieved from other agents in any case, or only when no local relevant plans are available, respectively.
- `acquisitionPolicy(Te,Acquisition)`, where *Acquisition* may assume the values `discard`, `add` and `replace` meaning that, when a relevant plan for *Te* is retrieved from a trusted agent, it must be used and discarded, or added to the plan library, or used to update the plan library by replacing all the plans triggered by *Te*.

Plans

In this extension plans also have

- a **source** which determines the first owner of the plan
- an **access specifier** which determines the set of agents with which the plan can be shared.

The **source** may assume two values:

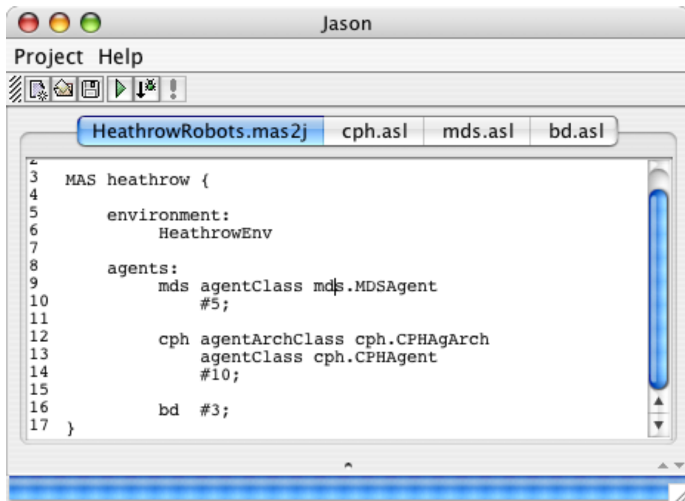
- `self` (the agent possesses the plan)
- `Ag` (the agent was originally from *Ag*).

The **access specifier** may assume three values:

- `private` (the plan cannot be shared),
- `public` (the plan can be shared with any agent) and
- `only(TrustedAgentSet)` (the plan can be shared only with the agents contained in *TrustedAgentSet*).

Main Features of the Jason Platform

Configuring Multi-Agent Systems



The BNF grammar in Figure ?? gives the syntax that can be used in the configuration file:

- **<NUMBER>** is used for integer numbers;
- **<ASID>** are AgentSpeak identifiers, which must start with a lowercase letter;
- **<ID>** is any identifier (as usual);
- **<PATH>** is as required for defining file pathnames as usual in ordinary operating systems;
- **<ID>** used after the keyword `MAS` is the name of the society;
- `architecture` is used to specify which of the two overall agent architectures available with **Jason**'s distribution will be used;
- `environment` needs to be referenced (Name of Java class for the environment);
- `agents` is used for defining the set of agents that will take part in the multi-agent system.

```
mas           → "MAS" <ID> "{"  
                [ "architecture" ":" <ID> ]  
  
                environment  
                agents  
                "}"  
environment → "environment" ":" <ID> [ "at" <ID> ]  
  
agents       → "agents" ":" ( agent )+
```

agent

→ **<ASID>**
 [filename]

 [options]

 ["agentArchClass" <ID>]

 ["agentClass" <ID>]

 ["#" **<NUMBER>**]

 ["at" **<ID>**]

";"

filename→ [**<PATH>**] **<ID>**options→ "[" option ("," option) * "]"

```
option      → "events" "=" ( "discard" | "requeue" | "retrieve" )  
  
            |  "intBels" "=" ( "sameFocus" | "newFocus" )  
  
            |  "verbose" "=" <NUMBER>
```

Figure 2: BNF of the Language for Configuring Multi-Agent Systems.

Settings available for the AgentSpeak interpreter:

- `events`: Options are either discard, `requeue`, or `retrieve`;
- `intBels`: Options are either sameFocus or `newFocus`;
- `verbose`: A number between 0 and 6 should be specified. The higher the number, the more information about that agent is printed out in the console where the system was run.

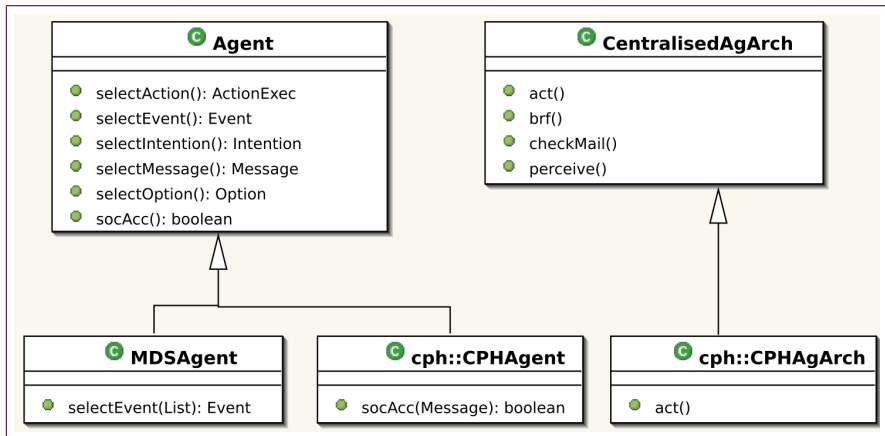
Creating Environments I

```
public class HeathrowEnv extends Environment {
    Map agsLocation = new HashMap();
    public List getPercepts(String agName) {
        if ( ... unattended luggage has been found ... ) {
            // all agents will perceive the fact that
            // there is unattendedLuggage
            getPercepts().add(Term.parse("unattendedLuggage"));
        }
        if (agName.startsWith("mds")) {
            // mds robots will also perceive their location
            List customPerception = new ArrayList();
            customPerception.addAll(getPercepts());
            customPerception.add(agsLocation.get(agName));
            return customPerception;
        } else {
            return getPercepts();
        }
    }
}
```

Creating Environments II

```
    }  
}  
public boolean executeAction(String ag, Term action) {  
    if (action.hasFunctor("disarm")) {  
        ... the code that implements the disarm action  
        ... on the environment goes here  
    } else if (action.hasFunctor("move")) {  
        ... the code for changing the agents' location and  
        ... updating the agsLocation map goes here  
    }  
    return true;  
}  
}
```


Customising Agents



Example 1.4 (Customised selectEvent)

The MDS79 robots must give priority to events related to unattended luggage over any other type of event. A customised MDS79 agent class:

```
public class MDSAgent extends Agent {
    public Event selectEvent(List evList) {
        Iterator i = evList.iterator();
        while (i.hasNext()) {
            Event e = (Event)i.next();
            if (e.getTrigger().getFunctor().equals(
                                                "unattendedLuggage")) {
                i.remove();
                return e;
            }
        }
        return super.selectEvent(evList);
    }
}
```

Example 1.5 (Act method)

Under no circumstances a CPH903 robot is allowed to disarm a bomb. Override the act method in the CPH903's customised AgArch class and ensure that the selected action is not `disarm` before allowing it to be executed in the environment:

```
public class CPHAgArch extends CentralisedAgArch {  
    public void act() {  
        // get the current action to be performed  
        Term action = fTS.getC().getAction().getActionTerm();  
  
        if ( \relax !action.getFunctor().equals("disarm") ) {  
            // ask the environment to execute the action  
            fEnv.executeAction(getName(), action);  
            ...  
        }  
    }  
}
```

Defining New Internal Actions

Example 1.6

```
package mds;
import ...
public class calculateMyBid implements InternalAction {
    public boolean execute(TransitionSystem ts, Unifier un,
                           Term[] args) throws Exception {
        int bid = ... a complex formula ...;
        ... plus complex algorithm and calculations
            for adjusting the agent's bid ...

        un.unifies(args[0], Term.parse(""+bid));
        return true;
    }
}
```

Example of AgentSpeak Plans for an Airport Security Robot

```
free. // I'm not currently handling unattended luggage

+unattendedLuggage(Terminal, Gate) : true
    <- !negotiate.

@pn1
+!negotiate : not free
    <- .broadcast(tell, bid(0)).
```

```
@pn2
+!negotiate : free
  <- .myName(I); // Jason internal action
  +winner(I); // belief I am the negotiation winner
  +bidsCount(1);
  mds.calculateMyBid(Bid); // user internal action
  +myBid(Bid);
  .broadcast(tell, bid(Bid)).

@pb1 // for a bid better than mine
+bid(B) [source(Sender)] :
    myBid(MyBid) & MyBid < B &
    .myName(I) & winner(I)
  <- -winner(I);
  +winner(Sender).
```

```
@pb2 // for other bids (and I'm still the winner)
+bid(B) : .myName(I) & winner(I)
    <- !addBidCounter;
    !endNegotiation.

@pend1 // all bids was received
+!endNegotiation : bidsCount(N) & numberOfMDS(M) & N >= M
    <- -free; // I'm no longer free
    !checkUnattendedLuggage.

@pend2 // void plan for endNegotiation not to fail
+!endNegotiation : true <- true.
:
:
```

Figure 3: Example of AgentSpeak Plans for an Airport Security Robot.

Available Tools and Documentation

Jason is distributed with an IDE which provides a GUI as well as AgentSpeak code. Through the IDE, it is also possible to control the execution of a MAS, and to distribute agents over a network in a very simple way. There are three execution modes:

- 1 Asynchronous (default)
- 2 Synchronous
- 3 Debugging

Jason's distribution comes with documentation which is also available online at <http://jason.sourceforge.net/Jason.pdf>.

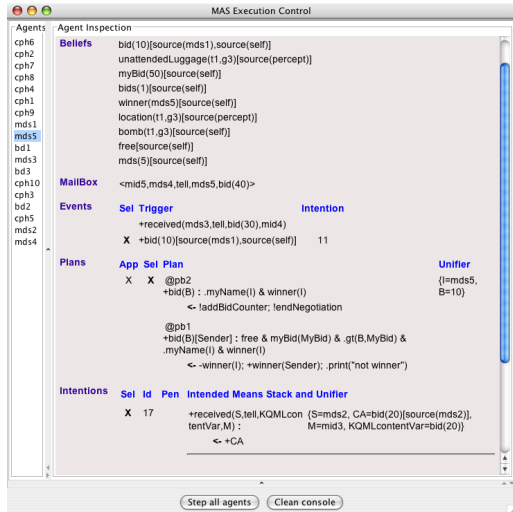


Figure 4: Jason's Mind Inspector.

Standards Compliance, Interoperability, and Portability

- As **Jason** is implemented in Java, there is no issue with portability.
- Components of the platform can be easily changed by the user.

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** **Jason** is based on a BDI logic-programming language and therefore fully supports all these concepts; it does not as yet support agent organisations, but there is ongoing work to support that in the future.

Communication

- Q:** Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A:** Speech-act based communication is available in **Jason**, based on KQML performatives and some extra ones that are used for exchanging plans (rather than beliefs).

Underlying Computational Model

Q: Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?

A: SACI supports strong mobility, but we have not as yet provided mobility within **Jason**; however, given that this is already available through SACI, it is straightforward to provide a standard internal action allowing AgentSpeak programmers to use mobility (this should be available in **Jason**'s next release).

Simplicity

Q: How easy it is to use and understand the language?

A: The core of **Jason** is an AgentSpeak interpreter, which is, in our opinion, the most simple and elegant, yet quite expressive, agent-oriented programming language that appears in the literature.

Preciseness

- Q: Does the language have clear and precise semantics? How has it been formalised?
- A: There is formal semantics for AgentSpeak with the main extensions available in **Jason**; the semantics was given using Plotkin's structural approach to operational semantics.

Expressiveness

- Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?
- A: **Jason** should be suitable for any application for which BDI agents are suitable (varied applications of such agents have appeared in the literature).

Extensiveness

- Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?
- A: The “internal action” construct allows for some form of extensibility, and there are various customisation mechanisms available in **Jason**.

Verification

Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?

A: Model checking techniques that apply directly on (a restricted version of) AgentSpeak are being developed by Bordini, Fisher, Wisser, and Wooldridge. To our knowledge, this is the only agent programming language for which work on model checking techniques has been published.

Software Engineering Principles

- Q: Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?
- A: Very little has been considered in this area as yet; however, methodologies suitable for BDI-like agents, such as Prometheus (by Padgham and Winikoff), should be suitable for implementation with **Jason**.

Language Integration

- Q:** Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?
- A:** The same “internal action” construct mentioned above allows for a high-level approach to integration with Java (the agent code itself remains a clear logical description of the agent’s reasoning, as Java or legacy code is simply referenced in the high-level internal actions).

Language Integration (continued)

- Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?
- A: This can be achieved by the use of JNI (Java Native Interface) and the internal action mechanism mentioned above.

Platform

Deployment and Portability

- Q: Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A: **Jason** has sufficient documentation, and further tutorials are under preparation; when agents are not situated in some real-world environment, deployment in a networked system is trivial with **Jason** (through the use of SACI).

Deployment and Portability (continued)

Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?

A: No, it runs on any platform for which Java is available.

Standards Compliance

- Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?
- A: **Jason** provides these services through SACI, which is KQML compliant; however, the distribution infrastructure can be customised, so a different infrastructure (e.g., one that is FIPA compliant) can be used if necessary.

Platform Extensibility

Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?

A: **Jason** is available **Open Source** and in most extensions attempted so far it has proven very easy to extend (because of the customisation mechanisms, this often means that not change in the interpreter itself is required); there are a number of extensions planned for the near future.

Available Tools

- Q: What tools are provided by the platform for the management, monitoring, logging and debugging of applications?
- A: **Jason** has a debugging mode in which the system can be run step by step and a “mind inspector” which allows the user to check the mental attitudes of agents running across a network.

Available Tools (continued)

- Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?
- A: There is documentation which is partly a tutorial on AgentSpeak and partly a manual for using the platform; improvements on this documentation and further tutorials are expected.

Available Tools (continued)

Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?

A: **Jason** comes with an IDE which is very simple to use.

Tool Integration

- Q:** In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?
- A:** Applications developed with **Jason** have not made use of integrated tools, but in principle any tool that integrates satisfactorily with Java should integrate with **Jason** as well.

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** Applications require a Java Virtual Machine to run, but there are no operating system requirements.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: **Jason** has changed significantly in the last year, and we have not yet updated such statistics, but we plan to include such figures in the manual in future releases.

Performance Issues (continued)

- Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?
- A: We have recently released version 0.6 (open source, as usual); although there are stable versions, various significant changes have been made from each version to the next (Jason is very much work on progress).

Multi-Agent Systems Features

Q: Does the platform support open multi-agent systems and heterogeneous agents?

A: Again through the use of SACI, open multi-agent systems are easily supported; although heterogeneity is in principle possible, various features (e.g., plan exchange) still consider that the all agents are developed in AgentSpeak.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: Both centralised and distributed execution is available; social structures are still not currently supported, but this is ongoing work.

Multi-Agent Systems Features (continued)

Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?

A: This is not as yet available, but certainly planned for the future.

Applications Supported by the Language and Platform

Typical Examples

- Q: What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?
- A: So far, apart from academic coursework, **Jason** has only been used for social simulation and autonomous characters for computer animation.

Targeted Domains

Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?

A: BDI agents are suitable for a variety of domains; we are particularly interested in Semantic Web and Grid-based applications; specifically, we plan to develop Grid-based social simulations in the near future.

References I



Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini. *Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange*. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004)*, New York, NY, 19–23 July, pages 698–705, New York, NY, 2004. ACM Press.



Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge.

Model checking AgentSpeak.

In Jeffrey S. Rosenschein, Tuomas Sandholm, Michael Wooldridge, and Makoto Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003)*, Melbourne, Australia, 14–18 July, pages 409–416, New York, NY, 2003. ACM Press.

References II



Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge.
Model checking rational agents.

IEEE Intelligent Systems, 19(5):46–52, September/October 2004.



Rafael H. Bordini, Jomi F. Hübner, et al.

Jason: *A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, version 0.6 edition, Feb 2005.

<http://jason.sourceforge.net/>.



Rafael H. Bordini, Fabio Y. Okuyama, Denise de Oliveira, Guilherme Drehmer, and Romulo C. Krafta.

The MAS-SOC approach to multi-agent based simulation.

In Gabriela Lindemann, Daniel Moldt, and Mario Paolucci, editors,

Proceedings of the First International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), 16 July, 2002, Bologna, Italy (held with AAMAS02) — Revised Selected and

References III

Invited Papers, number 2934 in Lecture Notes in Artificial Intelligence, pages 70–91, Berlin, 2004. Springer-Verlag.



Rafael H. Bordini, Willem Visser, Michael Fisher, Carmen Pardavila, and Michael Wooldridge.

Model checking multi-agent programs with CASP.

In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of the Fifteenth Conference on Computer-Aided Verification (CAV-2003)*, Boulder, CO, 8–12 July, number 2725 in Lecture Notes in Computer Science, pages 110–113, Berlin, 2003. Springer-Verlag.

Tool description.

References IV



Rodrigo Machado and Rafael H. Bordini.

Running AgentSpeak(L) agents on SIM_AGENT.

In John-Jules Meyer and Milind Tambe, editors, *Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, number 2333 in Lecture Notes in Artificial Intelligence, pages 158–174, Berlin, 2002. Springer-Verlag.



Álvaro F. Moreira, Renata Vieira, and Rafael H. Bordini.

Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication.

In João Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies, Proceedings of the First International Workshop (DAL-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Revised Selected and Invited Papers)*, number 2990

References V

in Lecture Notes in Artificial Intelligence, pages 135–154, Berlin, 2004. Springer-Verlag.



Anand S. Rao.

AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42–55, London, 1996. Springer-Verlag.



Anand S. Rao and Michael P. Georgeff.

Decision procedures for BDI logics.

Journal of Logic and Computation, 8(3):293–343, 1998.

References VI



Jorge A. Torres, Luciana P. Nedel, and Rafael H. Bordini.

Autonomous agents with multiple foci of attention in virtual environments.
In *Proceedings of 17th International Conference on Computer Animation and Social Agents (CASA 2004)*, Geneva, Switzerland, 7–9 July, pages 189–196, 2004.

Abstract

This chapter presents 3APL, which is a multi-agent programming language, and its corresponding development platform. The 3APL language is motivated by cognitive agent architectures and provides programming constructs to implement individual agents directly in terms of beliefs, goals, plans, actions, and practical reasoning rules. The syntax and semantics of the 3APL programming language is explained. Various features of the language and platform and some software engineering issues are discussed.

Keywords

Multi-Agent Programming Language, Cognitive Agents, Multi-Agent Systems

Authors

Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer

- Agents are often described using **logic**.
- Concepts that are commonly incorporated:
 - Knowledge;
 - Beliefs;
 - Desires;
 - Intentions;
 - Commitments;
 - Goals;
 - Plans.
- Using an arbitrary programming language it will be difficult to verify whether it satisfies its specification.
- \leadsto **3APL** ("triple-a-p-l") to support the **practical development** of intelligent agents.

3APL

Basic building blocks:

- **Beliefs**;
- **Plans**;
- **Rules** for revising plans.

Extensions

- **Declarative goals** (“goal”) describe the **state** an agent wants to reach and can be used to program pro-active behavior.
- **Communication** to allow describing multi-agent 3APL systems.

Issues

Given these mental attitudes, issues arise:

- which plan should be executed at a certain point;
- which goal(s) should be pursued;
- which (type of) rule should be applied;
- etc.

The choices made affect the operation of the agent and it is thus an important point to consider.

~→ **Meta-language** on top of basic 3APL.

Implementation of a multi-agent system

In general:

- One **single-agent** programming language to implement individual agents;
- One **multi-agent** programming language to implement multi-agent aspects.

Implementation of a multi-agent system I

In 3APL:

- A 3APL multi-agent system consists of a set of **concurrently** executed 3APL agents that can interact with each other either **directly** through **communication** or **indirectly** through the shared environment;
- To implement a 3APL multi-agent system, the 3APL platform has been built to support the design, implementation, and execution of a set of 3APL agents that share an external environment;
- All organization and coordination issues should be implemented **implicitly** through the implementations of **individual 3APL agents**.

Implementation of a multi-agent system II

- Individual 3APL agents can be implemented by the 3APL programming language;
- Shared environment can be implemented in the Java programming language (Java class);
- Its methods correspond with the actions that agents can perform in the environment;
- Agents can interact with each other through direct communication.

3APL programming language

- **Mental attitudes** (data structures);
 - Beliefs;
 - Goals;
 - Plans;
 - Actions (mental, internal, communication);
 - Reasoning rules.
 - **Deliberation process** (programming instructions).
 - Selection and execution of actions and plans (for modifying an agent's belief base or the shared environment);
 - Selection and application of reasoning rules (for shifting goal and plan bases).
- The basic deliberation constructs can be composed by sequential composition, if-then-else and while constructs. \leadsto Language is expressive enough to implement important aspects of **subsumption architectures**.

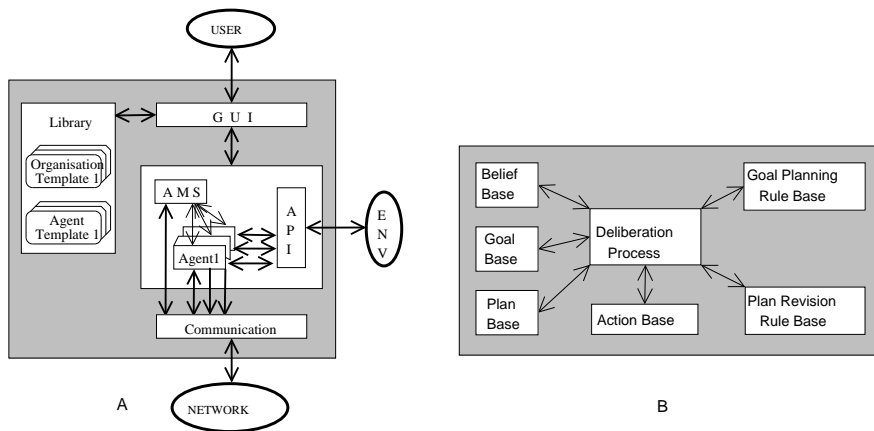


Figure 5: The architectures of 3APL platform (A) and individual 3APL agents (B)

The 3APL platform consists of:

- A **number of agents**;
- A **directory facilitator** called **agent management system (AMS)**;
- A **message transport system** which delivers messages between agents;
- A **shared environment**;
- A **plugin interface** that allows agents to execute actions in the shared environment.

Function of the agent management system (AMS)

- Register agents that are loaded and executed on the platform and it answers a set of questions from agents about other agents that are present on the platform.
- These questions can be, for example, about the names of agents, their functions, and the services they provide.

Each individual 3APL agent consists of:

- A belief base;
- A goal base;
- A plan base;
- An action base for the specification of internal mental actions;
- A base for goal planning rules (which can be applied to plan a goal)
- A base for plan revision rules (which can be used to revise, adopt, and drop plans).

Specifications and Syntactical Aspects

EBNF grammar of the language

We use:

- $\langle atom \rangle$ to denote an **atomic formula** the terms of which can include Prolog-like list representations of the form $[a, b, [3, f]]$, $[X|T]$, and $[a, [4, d] | T]$, etc.;
- $\langle ground_atom \rangle$ to denote a **ground atomic formula**, which is an atomic formula that contains no variables. The terms of ground atomic formulae can include Prolog-like list representations such as $[a, b, c]$, $[e, [9, d, g], 3]$;
- $\langle Atom \rangle$ to denote atomic formulae where the predicate letter starts with a capital letter;
- $\langle ident \rangle$ to denote a string;
- $\langle var \rangle$ to denote a variable.

Specifications and Syntactical Aspects

EBNF specification for individual agents I

```
 $\langle Program \rangle ::=$  "Program"  $\langle ident \rangle$   
    ( "Load"  $\langle ident \rangle$  )?  
    "Capabilities:" (  $\langle capabilities \rangle$  )?  
    "BeliefBase:" (  $\langle beliefs \rangle$  )?  
    "GoalBase:" (  $\langle goals \rangle$  )?  
    "PlanBase:" (  $\langle plans \rangle$  )?  
    "PG-rules:" (  $\langle p\_rules \rangle$  )?  
    "PR-rules:" (  $\langle r\_rules \rangle$  )?
```

Specifications and Syntactical Aspects

EBNF specification for individual agents II

```

<capabilities> ::= <capability> ( "," <capability> ) *
<capability>  ::= " { " <query> " } " <Atom> " { " <literals> " } "
<beliefs>     ::= ( <belief> ) *
<belief>      ::= <ground_atom> "." |
                  <atom> ":" — " <literals> "."
<goals>       ::= <goal> ( "," <goal> ) *
<goal>        ::= <ground_atom> ( "and" <ground_atom> ) *
<plans>       ::= <plan> ( "," <plan> ) *
<plan>        ::= <basicaction> | <composedplan>

```

Specifications and Syntactical Aspects

EBNF specification for individual agents III

```
 $\langle basicaction \rangle ::= \text{"}\epsilon\text{"} \mid \langle Atom \rangle \mid$ 
```

```
                  \text{"Send("}\langle iv \rangle, \langle iv \rangle, \langle atom \rangle\text{"} \mid
```

```
                  \text{"Java("}\langle ident \rangle, \langle atom \rangle, \langle var \rangle\text{"} \mid
```

```
                  \langle wff \rangle \text{"?"} \mid \langle atom \rangle
```

```
 $\langle composedplan \rangle ::= \text{"if"} \langle wff \rangle \text{"then"} \langle plan \rangle$ 
```

```
                  ( \text{"else"} \langle plan \rangle )? \mid
```

```
                  \text{"while"} \langle query \rangle \text{"do"} \langle plan \rangle \mid
```

```
                  \langle plan \rangle \text{";" } \langle plan \rangle
```

Specifications and Syntactical Aspects

EBNF specification for individual agents IV

```

<p_rules> ::= <p_rule> ( "," <p_rule> )*
<p_rule>  ::= <atom> "<->" <query> "|" <plan>
<p_rule>  ::= "<->" <query> "|" <plan>
<r_rules> ::= <r_rule> ( "," <r_rule> )*
<r_rule>  ::= <plan> "<->" <query> "|" <plan>
<literals> ::= <literal> ( "," <literal> )*
<literal>  ::= <atom> | "not("<atom>")"
<wff>      ::= <literal> | <wff> "and" <wff> |
               <wff> "or" <wff>
<query>    ::= <wff> | "true"
<iv>       ::= <ident> | <var>
  
```

Beliefs and goals

- The **beliefs** of a 3APL agent describe the situation the agent is in.
 - Beliefs are implemented by the **belief base**, which contains information the agent believes about the world.
 - The **goals** of the agent denote the situation the agent wants to realize, which is implemented by an agent's **goal base**.
-
- The belief base is implemented by a Prolog program consisting of Prolog facts and rules.
 - The initial belief base of a 3APL agent is preceded by the keyword `"BeliefBase:"`.
 - Note that the syntax of Prolog is in accordance with the specification of $\langle \textit{beliefs} \rangle$ as given above.

Example 2.1 (Initial belief base of a 3APL agent)

Blocks *a* and *b* are on the floor, block *c* is on block *a*, and that a block is clear if there is no block placed on top of it.

BeliefBase:

`on(a, fl) .`

`on(b, fl) .`

`on(c, a) .`

`clear(Y) :- not (on(X, Y)) .`

The specification of beliefs allows the use of negation in the body of the rules. The `not` in these rules stands for negation-as-failure.

- The **goal base** of a 3APL agent is a set of goals, each of which is implemented by a conjunction of ground Prolog atoms.
- The **initial goal base** of a 3APL agent is preceded by the keyword `"GoalBase :"`.

Example 2.2 (Initial goal base of a 3APL agent)

The agent has two goals. The first goal is to have block `a` on block `b` and block `b` on block `c`, and the second goal is to have block `d` on the floor.

`GoalBase:`

`on(a,b) and on(b,c) , on(d,fl)`

Two goals and the single goal `on(a,b) and on(b,c) and on(d,fl)` differ!

Specifications and Syntactical Aspects

Source from a formula

It is useful to be able to check whether a formula follows from:

- The belief base;
- The goal base.

Useful for:

- Test actions;
- Application of reasoning rules;
- Performing mental actions.

Belief and goal query expressions

For these purposes, we use the so-called **belief and goal query expressions** (i.e. $\langle query \rangle$) which are:

- Either the special atomic formulae `true`;
 - Or a well-formed formula (i.e. $\langle wff \rangle$) constructed from atoms and logical connectors.
-
- The keywords `and`, `or`, and `not` are used as logical connectives.

Example 2.3

$(\text{on}(X,b) \text{ and } \text{on}(b,Y)) \text{ or } \text{not}(\text{on}(b,fl))$ can be a belief query expression which is derivable from the belief base if:

- Either $\text{on}(X,b) \text{ and } \text{on}(b,Y)$ is derivable from the belief base;
- or $\text{on}(b,fl)$ is not derivable.

(Because of Prolog the `or` and `and` operators are not commutative.)

Specifications and Syntactical Aspects

Basic Actions

- In order to reach its goals, a 3APL agent adopts plans.
- A plan is built from basic actions that can be composed through co-called program operators.
- Five **basic actions** + **neutral action** (denoted by ϵ)

Basic Actions

Various kinds of basic actions:

- Mental actions;
- Communication actions;
- External actions;
- Test actions;
- Abstract plans.

Mental actions

A **mental action**:

- Can update the belief base of agents, if successfully executed;
- Has the form of an atomic formula
(**predicate name (i.e. $\langle Atom \rangle$) + list of terms**);
- Consists of
 - **Effects** (post-condition)
 - **Conditions** (pre-condition)

Capabilities

The **pre- and post-conditions** of **mental actions** are specified through so-called **capabilities** which consist of three parts:

- The mental action itself (i.e. $\langle Atom \rangle$);
 - A pre-condition which is a belief query expression (i.e. $\langle query \rangle$);
 - A post-condition which is a list of literals (i.e. $\langle literals \rangle$).
-
- An agent can execute a mental action if the pre-condition of the corresponding capability holds.
 - The effect of the execution of a mental action is then a change in the agent's belief base such that the post-condition of the corresponding capability holds.

In order to realize this effect, a function is defined in the interpreter that:

- **Adds the positive literals** to the **belief base**;
- **Retracts the atoms of the negative literals** from the **belief base**.

In the implementation of 3APL, the specification of capabilities is preceded by the keyword `"Capabilities:"`

Example 2.4 (Mental action)

The following is an example of a capability that defines the effect of the mental **Move** action.

Capabilities:

$\{ \text{on}(X, Y) \}$ **Move** (X, Y, Z) $\{ \text{not}(\text{on}(X, Y)) , \text{on}(X, Z) \}$

- Action **Move** (X, Y, Z) moves a block X from block Y to block Z .
- If this **Move** (X, Y, Z) action is executed, the variables X , Y and Z will be instantiated with a value.
- Assume for example that $X = a$, $Y = b$ and $Z = c$
 - The action can then be executed in case $\text{on}(a, b)$ is derivable from the belief base, i.e., if block a is on b .
 - The result should be that $\text{not}(\text{on}(a, b))$ and $\text{on}(a, c)$ are derivable from the belief base.
 - This is implemented by removing fact $\text{on}(a, b)$ and adding $\text{on}(a, c)$.

Send action

- A **send action** can be used to pass a message to another agent.
- A **message** contains:
 - The **name of the receiver** of the message;
 - The **speech act or performative (e.g. inform, request, etc.)**;
 - The content.
- The send action is like an **atomic formula** which has `Send` as the predicate name and has **three arguments**:
 - 1 Identifier or variable (i.e. $\langle iv \rangle$).
Name of the receiving agent;
 - 2 Identifier or variable (i.e. $\langle iv \rangle$).
Performative of the message;
 - 3 Atomic formula (i.e. $\langle atom \rangle$).
Content of the message.

Example 2.5 (Send action)

An example of a send action is `Send(ag2, inform, on(a,b))`, which specifies that agent `ag1` informs agent `ag2` that block `a` is on block `b`.

If an agent sends a message `Send(Receiver, Performative, Content)` to another agent:

- The belief base of the sender is updated with the formula `sent(Receiver, Performative, Content);`
- The belief base of the receiver is updated with the formula `received(Sender, Performative, Content).`

Agents can **receive a message** in their belief base at **each moment in time**. Note that **unlike the mental actions**, the **send actions can always be executed**.

External actions

- The **external actions** are means to change the external environment in which the agents operate.
- The **effects** of external actions are assumed to be determined by the environment and might not be known to the agents.
- The **agent** thus **decides to perform an external action** and the **external environment determines** the **effect** of this action.
- The agent can come to know the effects of an external action by performing a **sense action**.
- This **sense action** can be defined as an external action in an agent's plan, or it could be a pre-defined operation that is part of the sense-reason-act loop of the agent's deliberation cycle.

- External actions are performed by 3APL agents with respect to an environment which is assumed to be implemented as a Java class.
- In particular, the actions that can be performed in this environment are determined by the methods of the Java class (i.e., the methods specify the effect of those actions in that environment).
- The state of the environment is represented by the instance variables of the class.

- The external actions that can be performed by 3APL agents have the form `Java(Classname, Method, List)` where:
 - `Classname` is the name of the Java class that implements the environment;
 - `Method` is the action to be performed in the environment;
 - `List` is a list of returned values.
- The parameter `Method` corresponds with a parameterized method of the Java class `Classname`.
- `List` is a list of values returned by `Method`.
- The method can be implemented to return the result of the action in the list, or the list could for example be empty.
- In that case, an explicit sense action would have to be executed to obtain the result of the action.

Example 2.6 (External action)

- An example of an external action is
`Java(BlockWorld, east(), L)`
where the external action `east()` is performed in the environment `BlockWorld`.
- The effect of this action is that the position of the agent in the block world environment is shifted one slot to the east.

Test action

- A **test action** checks whether a **well-formed formula** (i.e. $\langle wff \rangle$) is derivable from the belief base.
- Such an action, which consists of a **well-formed formula** followed by a **question mark**, will be blocked if the formula is not derivable from the belief base.

Note that the derivation relation is implemented by the Prolog reasoning engine.

If the arguments of a test action are variables and the well-formed formula is derivable from the belief base, then the effect of the test action is a substitution that assigns terms to the variables.

The assignment is useful for retrieving information from the belief base and passing it to other actions for further manipulation.

Example 2.7 (Test action)

- An example of a **test action** is $(\text{on}(a, X) \text{ and } \text{on}(X, c)) ?$ which will be successfully executed if the agent believes that there is a block X placed on top of block c such that block a is placed on top of it.
- The result of a successful execution is a substitution such as $\{X/b\}$ which indicates that the relevant block is block b .

Abstract plan

- An **abstract plan**, which is represented as an atomic formula (i.e. $\langle atom \rangle$), is an abstract representation of a plan which can be instantiated with a (more concrete) plan during execution.
- An abstract plan cannot be executed directly and should be rewritten into another plan, possibly (and even probably) containing executable basic actions, through application of reasoning rules.
- The application of rules to abstract plans involves a unification of abstract plans with the head of rules through which values can be passed to the instantiated plan.

Specifications and Syntactical Aspects

Plans

Basic actions can be composed to build plans through so-called program operators.

Plans

There are three 3APL program operators:

- The sequential operator (denoted by `;`);
- The iteration operator (denoted by a `while-do` construct);
- The conditional choice operator (denoted by an `if-then-else` construct).

Definition 2.8 (Plans)

If β is a well-formed formula, β' is a query expression (i.e. a well-formed formula or `true`), and *Actions* is the set of basic actions as defined above, then the set of plans, denoted by *Plans* is defined as follows:

- $Actions \subseteq Plans$
- if $\pi, \pi' \in Plans$, then $\text{if } \beta \text{ then } \pi \text{ else } \pi' \in Plans$
- if $\pi \in Plans$, then $\text{while } \beta' \text{ do } \pi \in Plans$
- if $\pi, \pi' \in Plans$, then $\pi; \pi' \in Plans$

We use ε to denote the empty plan and we identify $\varepsilon; \pi$ with π .

- The plan base of a 3APL agent consists of a set of plans.
- In 3APL the specification of the initial plan base of an agent is preceded by the keyword "PlanBase:" and consists of a number of plans separated by a comma.

Example 2.9 (Initial plan base)

The initial plan base of a 3APL agent:

```
PlanBase:
```

```
while (on(X,fl) and not(on(V,X)) do {  
    (on(Y,Z) and not(Z==fl));  
    Move(X,fl,Y)  
}
```

This plan base consists of one plan which will find all free blocks (blocks with no block on top) that are placed on the floor and move them to an existing block which itself is not placed on the floor.

Specifications and Syntactical Aspects

Reasoning Rules

In order to reason with goals and plans, 3APL has two types of rules:

- Goal planning rules;
- Plan revision rules.

These rules are conditionalized by beliefs.

Definition 2.10 (Goal planning rules (PG) and the set of plan revision rules (PR))

Let β be a query expression, κ be an atomic formula, and π, π_h, π_b be plans. The set of goal planning rules (PG) and the set of plan revision rules (PR) are then defined as follows:

$$\begin{aligned} \kappa \leftarrow \beta \mid \pi, \leftarrow \beta \mid \pi &\in PG \\ \pi_h \leftarrow \beta \mid \pi_b &\in PR. \end{aligned}$$

Goal planning rules

- The **goal planning rules** are used to generate plans to achieve goals.
- In the first goal planning rule, the belief condition β indicates when the plan π could be generated to achieve the specified goal κ .
- The second goal planning rules can be used to model reactive behavior by omitting the head of the rule.
- This special kind of goal planning rule states that under the belief condition β , a plan can be adopted.
- The specification of the set of goal planning rules is preceded by the keyword "PG – rules :".

Example 2.11 (goal planning rule)

The specification of a goal planning rule of a 3APL agent:

PG-rules:

$$\text{on}(X, Z) \leftarrow \text{on}(X, Y) \quad | \quad \text{Move}(X, Y, Z)$$

This rule states that if the agent wants to have block X on block Z , but it believes that X is on block Y , then it plans to move X from Y onto Z .

Plan revision rules

The **plan revision rules** are used to revise plans from the plan base. The specification of the set of plan revision rules is preceded by the keyword "PR-rules:".

Example 2.12 (Plan revision rules)

The specification of a plan revision rule of a 3APL agent:

PR-rules:

```
Move(X,Y,Z) ← not(clear(X))    |  
                                on(U,X)?;Move(U,X,fl);Move(X,Y,Z)
```

This plan revision rule informally means that if the agent plans to move block *X* from block *Y* onto block *Z*, but it cannot move *X* because (it believes that) there is a block on *X*, then the agent should revise its plan by finding out which block (*U*) is on *X*, moving *U* onto the floor, and finally moving *X* from *Y* onto *Z*.

Plan revision rules (Continued)

A plan revision rule $\pi_h \leftarrow \beta \mid \pi_b$ can be applied to a plan π , if π_h can be matched to a prefix of π , i.e., if π is of the form $\pi_h; \pi'$.

Example 2.13

A plan $\text{Move}(a, b, c); \text{Move}(b, fl, a)$ can be revised into a plan $\text{Move}(a, b, fl); \text{Move}(b, fl, a)$ by applying the plan revision rule $\text{Move}(a, b, c) \leftarrow \text{true} \mid \text{Move}(a, b, fl)$.

Note that a plan revision rule could be used to drop (part of) a plan if its body π_b is the empty plan ϵ .

Specifications and Syntactical Aspects

Deliberation Cycle

- The beliefs, goals, plans and reasoning rules form the mental attitudes or data structures of 3APL agents.
- These data structures can be modified by deliberation operations such as applying a rule or executing a plan.
- These deliberation operations constitute the deliberation process of individual agents.
- The deliberation process or program can be viewed as the interpreter, as it determines which deliberation operations should be performed in which order.

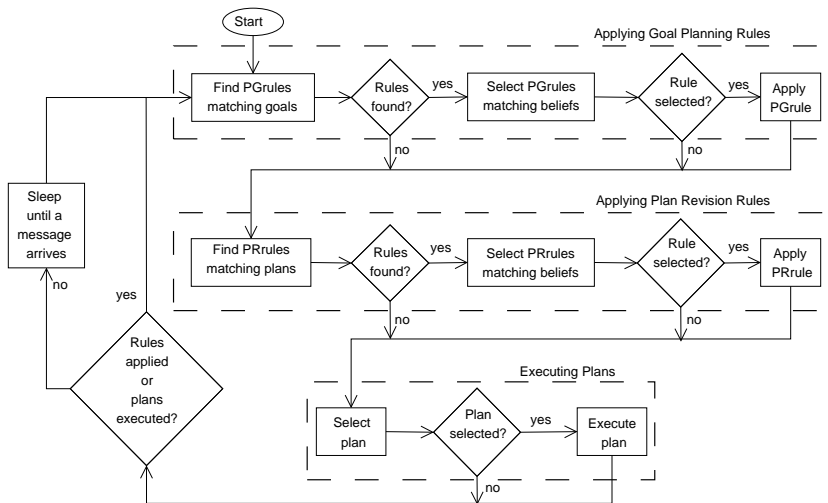


Figure 6: A cyclic interpreter (deliberation cycle) for the 3APL agents.

Specifications and Syntactical Aspects

3APL Platform

- The 3APL platform provides a user interface that allows 3APL agents to be programmed, loaded, and executed.
- During execution there are various facilities in the interface such as the sniffer, which allows monitoring the exchanges of messages between agents, and specific windows, which allow monitoring the changes of all mental attitudes of individual agents.
- There are various icons in the interface that allow monitoring the execution of agents, either step by step or continuously.
- The only part of the platform architecture that is programmable to this date is the shared environment.

Semantics and Verification

Definition 2.14 (Individual 3APL agent)

An 3APL agent is a tuple $\langle \mathfrak{t}, \sigma_0, \gamma_0, Cap, \Pi_0, PG, PR, \xi \rangle$ where:

- \mathfrak{t} is the agent identifier;
- σ_0 is the initial belief base;
- γ_0 is the initial goal base;
- Cap is the capability base;
- $\Pi_0 \subseteq Plans \times \{\text{true}\}$ is the initial plan base;
- PG is a set of goal planning rules;
- PR is a set of plan revision rules;
- ξ is the environment the agent shares with other agents; which is represented by a set of ground atoms.

Definition 2.15 ((ground) substitution, binding, domain, free variables)

- A substitution θ is a finite set of the form $\{x_1/t_1, \dots, x_n/t_n\}$, where $x_i \in Var$ and $t_i \in Term$ and $\forall i \neq j : x_i \neq x_j$.
- θ is called a ground substitution if all t_i are ground terms.
- Each element x_i/t_i is then called a binding for x_i .
- The set of variables $\{x_1, \dots, x_n\}$ is the domain of θ and will be denoted by $dom(\theta)$.
- The application of a substitution θ to a syntactic expression e is denoted as $e\theta$.
- It refers to the expression resulting from simultaneously replacing all occurrences of variable x in e for which $x/t \in \theta$ by t .

Definition 2.16 (Configuration)

A **configuration of an individual 3APL agent** is a tuple $\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$, where:

- ι is an **agent identifier**;
- σ is the **belief base** of the agent;
- γ is the **goal base** of the agent;
- Π is the **plan base** of the agent;
- θ is a **ground substitution** that binds domain variables to domain terms;
- and ξ is the **environment** it interacts with, where ξ is a set of ground atoms.

The goal base in a configuration is such that for any goal $\phi \in \gamma$ it holds that ϕ is not entailed by the agent's beliefs.

Definition 2.17 (Configuration (continued))

A **configuration of a 3APL multi-agent system** is a tuple $\langle \mathcal{A}_1, \dots, \mathcal{A}_n, \xi \rangle$ where:

- \mathcal{A}_i for $1 \leq i \leq n$ is the **configuration of individual agent i** ;
- and ξ is the **shared environment**.

This shared environment is the same as the environment of each individual agent.

The rationale behind the condition on the goal base is the following:

- The beliefs of an agent describe the state the agent is in and the goals describe the state the agent wants to realize;
- If an agent believes ϕ is the case, it cannot have the goal to achieve ϕ , because the state of affairs ϕ is already realized;
- This is thus an implementation of achievement goals, as opposed to maintenance goals.

Semantics and Verification

Transition system

- We define first a **derivation rule for transitions between multi-agent configurations**.
- This derivation rule, which captures the parallel execution of the set of individual agents, forms the only transition at the multi-agent level.

Definition 2.18 (multi-agent execution)

- Let $\mathcal{A}_1, \dots, \mathcal{A}_i, \dots, \mathcal{A}_n, \mathcal{A}'_i$ be agent configurations;
- let ξ and ξ' be specifications of the environment;
- let $\mathcal{A}_i = \langle \sigma, \gamma, \Pi, \theta, \xi \rangle$;
- let $\mathcal{A}'_i = \langle \sigma', \gamma', \Pi', \theta', \xi' \rangle$.

Then the derivation rule for multi-agent configurations is defined as:

$$\frac{\mathcal{A}_i \rightarrow \mathcal{A}'_i}{\langle \{\mathcal{A}_1, \dots, \mathcal{A}_i, \dots, \mathcal{A}_n\}, \xi \rangle \rightarrow \langle \{\mathcal{A}_1, \dots, \mathcal{A}'_i, \dots, \mathcal{A}_n\}, \xi' \rangle}$$

This derivation rule states that a transition between multi-agent configurations can be defined in terms of a transition between single-agent configurations. This amounts to an **interleaved** execution of the agents in the system.

- We now define transition rules that can derive transitions transforming single-agent configurations.
- These derivation rules specify the semantics of the execution of plans and the application of reasoning rules.
- The first derivation rule specifies the execution of the plan base of a 3APL agent.
- The plan base of the agent is a set of plan-goal pairs.
- This set can be executed by executing one of the constituent plans.
- The execution of a plan can change the agent's configuration.

Definition 2.19 (plan base execution)

Let

- $\Pi = \{(\pi_1, \kappa_1), \dots, (\pi_i, \kappa_i), \dots, (\pi_n, \kappa_n)\}$ and $\Pi' = \{(\pi_1, \kappa_1), \dots, (\pi'_i, \kappa_i), \dots, (\pi_n, \kappa_n)\}$ be plan bases;
- θ, θ' be ground substitutions;
- ξ, ξ' be environment specifications.

Then, the derivation rule for the execution of a set of plans is specified in terms of the execution of individual plans as:

$$\frac{\langle \mathbf{l}, \sigma, \gamma, \{(\pi_i, \kappa_i)\}, \theta, \xi \rangle \rightarrow \langle \mathbf{l}, \sigma', \gamma', \{(\pi'_i, \kappa_i)\}, \theta', \xi' \rangle}{\langle \mathbf{l}, \sigma, \gamma, \Pi, \theta, \xi \rangle \rightarrow \langle \mathbf{l}, \sigma', \gamma', \Pi', \theta', \xi' \rangle}$$

- An external action $\text{Java}(\text{Classname}, \alpha(t_1, \dots, t_n), x)$ has two functionalities:
 - Based on the input terms and the state of the environment, it generates a term (output of the action) and assigns it to variable x .
 - Actions are assumed to have effects on the environment.
- We assume:
 - For each external action with a method name α a function F_α which maps terms t_1, \dots, t_n and the environment ξ to a term which will be assigned to variable x .
 - A function G_α which maps terms t_1, \dots, t_n and the environment ξ to a new environment ξ' .
- An agent can execute an external action only if the goal associated to the action is still a goal of the agent.

Definition 2.20 (external action execution)

Let t, t_1, \dots, t_n be terms, x be a variable, let ξ, ξ' be agent environments, α be the method name of an external action, and assume functions F_α and G_α as explained above. The execution of an external action is then defined as follows:

$$\frac{\gamma \models \kappa}{\langle \mathbf{l}, \sigma, \gamma, (\text{Java}(\text{Classname}, \alpha(t_1, \dots, t_n), x), \kappa), \theta, \xi \rangle \rightarrow \langle \mathbf{l}, \sigma, \gamma, (\varepsilon, \kappa), \theta', \xi' \rangle}$$

where $\theta' = \theta \cup \{x/t\}$ with $t = F_\alpha(t_1, \dots, t_n, \xi)$, and $\xi' = G_\alpha(t_1, \dots, t_n, \xi)$.

Note that the execution of an external action thus influences only the substitution and the environment component of the configuration.

- The next type of basic action is the communication action $Send(r, p, \phi)$.
- We assume that each agent can receive a message at any moment in time.
- We use then a synchronization mechanism for sending and receiving messages.
- This synchronization mechanism takes care of simultaneously taking a message from the sending agent and putting it in the belief base of the receiving agent.
- How these messages are then handled by the receiving agent is done in a completely asynchronous fashion.

- The semantics of a $Send(r, p, \phi)$ action affects both sending and receiving agents.
- The communication action $Send(r, p, \phi)$ is removed from the plan base of the sending agent and the formula $sent(r, p, \phi)$ is added to its belief base.
- The formula $received(s, p, \phi)$ is added to the belief base of the receiving agent, where s is the name of the sending agent.
- This information about incoming and outgoing messages can respectively be used by the receiving and sending agents for their future deliberations.
- In order to be able to identify the sending agent when defining the addition of a fact of the form $received(s, p, \phi)$ to the belief base of the receiver, we add the name of the sending agent to messages.

Definition 2.21 (communication action execution)

Let $\langle s, r, p, \phi \rangle$ be the format of the message that is sent and received by the agents, where:

- s is the name of the sending agent;
- r is the name of the receiving agent;
- p is the communication performative;
- ϕ is the message content.

The following three transition rules specify the semantics for sending and receiving messages between agents, and their synchronization, respectively.

Definition 2.22 (communication action execution (continued))

The transition rule for the sending agent:

$$\frac{\gamma \models \kappa}{\langle s, \sigma, \gamma, (Send(r, p, \phi), \kappa), \theta, \xi \rangle \xrightarrow{\langle s, r, p, \phi \rangle!} \langle s, \sigma', \gamma, (\epsilon, \kappa), \theta, \xi \rangle}$$

where $\sigma' = \sigma \cup \{sent(r, p, \phi)\}$.

Definition 2.23 (communication action execution (continued))

The transition rule for the receiving agent:

$$\frac{}{\langle r, \sigma, \gamma, \Pi, \theta, \xi \rangle \xrightarrow{\langle s, r, p, \phi \rangle ?} \langle r, \sigma', \gamma, \Pi, \theta, \xi \rangle}$$

where $\sigma' = \sigma \cup \{received(s, p, \phi)\}$.

The transition rule guarantees that each agent can receive the messages that are directed to the agent at any moment in time.

Definition 2.24 (communication action execution (continued))

The transition rule for synchronization:

$$\frac{\langle \mathcal{A}_i, \xi \rangle \xrightarrow{\varphi?} \langle \mathcal{A}'_i, \xi \rangle, \langle \mathcal{A}_j, \xi \rangle \xrightarrow{\varphi!} \langle \mathcal{A}'_j, \xi \rangle}{\langle \{\mathcal{A}_1, \dots, \mathcal{A}_i, \dots, \mathcal{A}_j, \dots, \mathcal{A}_n\}, \xi \rangle \rightarrow \langle \{\mathcal{A}_1, \dots, \mathcal{A}'_i, \dots, \mathcal{A}'_j, \dots, \mathcal{A}_n\}, \xi \rangle}$$

- Next, we specify the derivation rule for the execution of the test action.
- A test action can bind the free variables that occur in the test formula for which no bindings have been computed yet.

Definition 2.25 (test execution)

Let β be a well-formed formula and let τ be a ground substitution.

$$\frac{\sigma \models \beta\theta\tau \ \& \ \gamma \models \kappa}{\langle \iota, \sigma, \gamma, \{(\beta?, \kappa)\}, \theta, \xi \rangle \rightarrow \langle \iota, \sigma, \gamma, \{(\varepsilon, \kappa)\}, \theta\tau, \xi \rangle}$$

- The entailment relation \models in the condition $\sigma \models \beta\theta\tau$ is implemented by the Prolog inference engine.
- When posing query β , substitution θ is first applied to β .
- The substitution τ is the substitution returned by Prolog and should bind the variables of $\beta\theta$.
- The entailment relation \models in $\gamma \models \kappa$ is implemented in a similar fashion.
- The derivation rules for the execution of composite plans are defined in a standard way.

- A **goal planning rule** $\kappa \leftarrow \beta \mid \pi$ specifies that the goal κ can be achieved by plan π if β is derivable from the agent's beliefs.
- A goal planning rule only affects the plan base of the agent.

Definition 2.26 (goal planning rule application)

Let $\kappa \leftarrow \beta \mid \pi$ be a goal planning rule. Let also τ_1, τ_2 be ground substitutions.

$$\frac{\gamma \models \kappa\tau_1 \ \& \ \sigma \models \beta\tau_1\tau_2}{\langle \mathbf{l}, \sigma, \gamma, \Pi, \theta, \xi \rangle \rightarrow \langle \mathbf{l}, \sigma, \gamma, \Pi \cup \{(\pi\tau_1\tau_2, \kappa\tau_1)\}, \theta, \xi \rangle}$$

- Note that the goal $\kappa\tau_1$ that should be achieved by the plan $\pi\tau_1\tau_2$ is associated with it.
- It is only this rule that associates goals with plans.
- The goal base of the agent does not change because the plan $\pi\tau_1\tau_2$ is not executed yet; the goals of agents may change only after execution of plans: goals are removed if believed to be achieved.
- We do not add substitutions τ_1, τ_2 to θ since these substitutions should only influence the new plan π .

The transition rule for the goal planning rule that defines reactive behavior, i.e. the goal planning rule in which the head is omitted, is a modification of the above transition rule.

Definition 2.27 (reactive goal planning rule application)

Let $\leftarrow \beta \mid \pi$ be a reactive goal planning rule and let also τ be a ground substitution.

$$\frac{\sigma \models \beta\tau}{\langle \mathbf{l}, \sigma, \gamma, \Pi, \theta, \xi \rangle \rightarrow \langle \mathbf{l}, \sigma, \gamma, \Pi \cup \{(\pi\tau, \text{true})\}, \theta, \xi \rangle}$$

Note that the goal associated to the generated plan is set to true, which means that the plan is not generated to achieve a specific goal.

Semantics and Verification

Semantics of a 3APL agent

- The semantics of an individual 3APL agent as well as the semantics of a 3APL multi-agent system is derived directly from the transition relation \rightarrow .
- The meaning of individual agents and multi-agent systems consists of a set of so called computation runs.

Definition 2.28 (computation run)

Given a transition system, a computation run $\text{CR}(s_0)$ is a finite or infinite sequence s_0, \dots, s_n or s_0, \dots where s_i are configurations, and $\forall_{i>0} : s_{i-1} \rightarrow s_i$ is a transition in the transition system.

We can now use the concept of a computation run to define the semantics of individual 3APL agents and the semantics of 3APL multi-agent systems.

Definition 2.29 (semantics of 3APL multi-agent systems)

The semantics of a 3APL multi-agent system $\langle \mathcal{A}_1, \dots, \mathcal{A}_n, \xi \rangle$ is the set of computation runs $\text{CR}(\langle \mathcal{A}_1, \dots, \mathcal{A}_n, \xi \rangle)$ of the transition system for 3APL multi-agent systems.

Note that the computation runs of a 3APL multi-agent system consist of multi-agent transitions which can be derived by means of two multi-agent transition rules. The first is defined in definition ?? and the second is the synchronization rule specified in definition ??.

Semantics and Verification

3APL Verification

In progress...

Software Engineering Issues

Separation of concerns

- Development methodologies for multi-agent systems differ from each other in many respects.
 - Some of them focus on inter-agent aspects;
 - while others also provide support for the design of internal components of an agent, such as mental attitudes and the deliberation process.
- Finally, some methodologies explicitly deal with the environment, while others do not.
- The tools to develop and implement multi-agent systems should therefore support each of these issues separately.

Software Engineering Issues

Separation of concerns I

- The 3APL programming language supports:
 - The implementation of inter-agent issues by providing the communication action **Send**;
 - The 3APL platform manages the transportation of the communicated messages.
- The platform provides information about existing agents to other agents through the AMS.
- The information provided by the AMS to agents is required for agents' interactions.
- The environment can be implemented directly and explicitly through external programs accessible to the agents through API's.

Software Engineering Issues

Separation of concerns II

- The 3APL programming language respects the separation of concerns related to the distinction between an agent's data structures and an agent's operations.
- The data structures are:
 - mental attitudes such as beliefs, goals, and plans;
 - operations concern manipulation of the mental attitudes such as updating of beliefs, plans and goals, and execution of plans.
- This distinction is made explicit by introducing two levels of programming:
 - At the data level one can specify the mental attitudes of the agents;
 - and at the operation level one can implement the deliberation process of the agent.

Software Engineering Issues

Modularity

The implementation of an agent is modular (seven different modules):

- **The capability base** of the agent which implements the mental actions that an agent can perform to update its beliefs;
- **The belief base** of the agent which contains information the agent believes about the world as well as information that is internal to the agent.
 - The first kind of initial beliefs constitutes the background knowledge which can be used by different agents;
 - The second kind of initial beliefs is specific to agents and cannot be used by other agents.

Software Engineering Issues

Modularity I

- The **goal base** that denotes the situation the agent wants to realize;
- The **plan base** of the agent which contains the plans that the agent intends to perform;
- The **goal planning rule base** that contains the rules that can be used to generate a plan for the possible goals of an agent;
- The **plan revision rule base** that contains rules to revise existing agent's plans;
- The **deliberation module** that allows the implementation of an agent's deliberation process.

Software Engineering Issues

Abstraction

- The abstraction mechanisms are related to external actions and abstract plans.
- The external actions allow users to use external programs through their corresponding API's without having any access to the internal data and operations of the programs.
- The 2. abstraction mechanism is related to abstract plans which allow users to abstract over certain parts of plans.
- The abstract plans can be instantiated with a plan through the application of plan revision rules.
- Note: An abstract plan should be introduced, not only because it occurs in different plans, but also because its specific instantiation depends on the conditions known only at run time

Example 2.30

- Going to work can be considered an abstract plan since its specific instantiations such as going to work by bus, by taxi, by train, or by own car depend on the conditions that hold when the plan is to be executed.
- If the agent does not have enough money, then it may consider going by bus or train, otherwise it may consider using a taxi.

- The introduction of abstract plans in 3APL implies the introduction of plan revision rules.
- In implementing 3APL agents, the programmers tend to conceive abstract plans as a kind of procedure calls and the plan revision rules as the corresponding procedure.
- It is important to note that this is not the optimal and principal use of abstract plans and their corresponding plan revision rules.

Software Engineering Issues

Reusability

- The 3APL platform allows reusing multi-agent systems by providing a library of templates for individual agents and templates for multi-agent systems.
- Using the templates for individual agents, the 3APL programmer can use generic agents that have certain initial mental attitudes.
- The templates for multi-agent systems, also known as projects, allow the 3APL programmers to use a set of generic agents that, in addition to their initial mental attitudes, follow a specified interaction protocol.
- Such a template can include an environment with which the agents are supposed to interact.

Example 2.31

An example of a multi-agent template is a template for an auction:

- In order to implement such an auction, a 3APL programmer can load such a multi-agent template and implement both the details of the agents, such as their specific initial mental attitudes, as well as the details of their environment.

Language integration

- The 3APL programming language together with its platform allows the integration of Prolog and Java.
- The Prolog programs can be integrated since they can be loaded in 3APL and used as background knowledge.
- Given a loaded Prolog program, the agent can pose queries in three different contexts:
 - as the pre-condition of mental actions;
 - as test actions in plans;
 - as the guard of the reasoning rules.

- The Prolog programs can thus be used to control the execution of mental actions, the execution of plans, and the application of reasoning rules.
- Note that the queries may yield substitutions that can bind other variables used in the post-conditions of the mental actions, in the rest of plans that follow a test action, and in the bodies of reasoning rules.

- The 3APL programming language allows Java programs to be used through external actions.
- The external actions can be used to call methods of Java classes.
- Using the arguments of these methods, it is possible to pass data from 3APL to Java and vice versa.
- In this way, data can be passed from Java to the plans of the agent to the Prolog part (belief base) of the agent and vice versa.
- Note that the integration of Java is also used to implement the multi-agent environment with which the agents interact.

Available tools and documentation

The 3APL platform is an experimental tool, designed to support the development, implementation, and execution of 3APL agents.

- Detailed information:

<http://www.cs.uu.nl/3apl/download/java/userguide.pdf>

- Tutorial and training material:

<http://www.cs.uu.nl/3apl>

- The implementation documentation of the platform:

<http://www.cs.uu.nl/3apl/docs/aplp-refman/index.html>

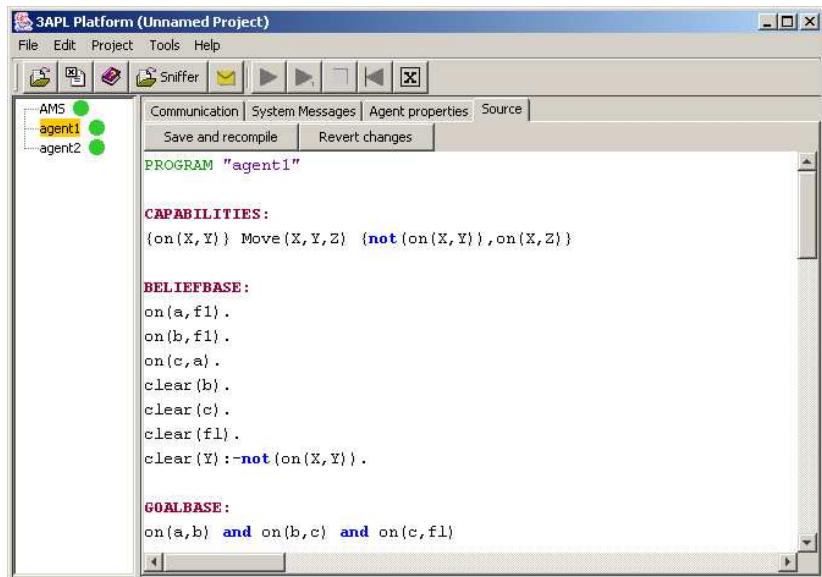


Figure 7: Graphical user interface of the 3APL platform.

Standards compliance, interoperability and portability

- The 3APL platform has been tested on Windows 98, Windows NT and Windows XP, as well as on Linux, Unix (Solaris) and Mac OS X.
- 3APL is written in Java 2 SDK 1.4, and makes use of the Prolog engine of JIProlog, which is also implemented in Java.
- The downloadable 3APL package consists of a .jar file that contains all the .class files needed, as well as examples of 3APL programs.
- The platform provides distributed control such that the agents can be executed concurrently.

Standards compliance, interoperability and portability I

- The platform also provides the possibility to build a library of agents, multi-agent systems and agent templates.
- The templates can be loaded and extended to build multi-agent systems.
- Based on the templates it is possible to have interaction protocols in the platform's library, since the protocols can be defined in terms of a set of agent templates in which only the actions prescribed by the protocols are specified.

- The applications that can be developed using the 3APL platform and the 3APL programming language are those that are best understood in terms of cognitive and social concepts like beliefs, goals, plans, actions, norms, organizational structures, resources and services that are part of the multi-agent environment.
- We have already implemented a number of toy problem applications such as block world logistics, Axelrod's tournament, English Auction, and Contract Net protocols.
- Also, 3APL is already applied to implement the high-level control of mobile robots.

- In this project, external actions of 3APL were defined and connected to some simple sensory and motor actions of the mobile robot.
- In this way, a programmer can implement a 3APL program that senses the position of the robot it is controlling and determine how to reach a goal position in a rectangular environment, a model of which is accessible to the 3APL program.
- Currently, 3APL is also being applied to control the behavior of SONY AIBO robots and to implement small device mobile applications.

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** The language supports the implementation of mental attitudes (beliefs, goals, plans, and reasoning rules), the implementation of a deliberation cycle, and reactive as well as deliberative behavior.

Communication

- Q:** Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A:** The language provides a speech-act based programming construct (the `Send` operator) for communication.

Underlying Computational Model

Q: Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?

A: The language does not support the design of mobile agents.

Simplicity

Q: How easy it is to use and understand the language?

A: The general ideas of the language can be understood relatively easily, especially for someone familiar with the idea of cognitive agents, as a limited number of language constructs is available. The details of the formal semantics will take some more time to comprehend.

Preciseness

Q: Does the language have clear and precise semantics? How has it been formalised?

A: The language has a clear and formal semantics, for the most part defined by means of a transition system.

Expressiveness

- Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?
- A: The applications that can be developed using the 3APL platform and the 3APL programming language are those that are best understood in terms of cognitive and social concepts like beliefs, goals, plans, actions, norms, organizational structures, resources and services that are part of the multi-agent environment.

Extensiveness

- Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?
- A: The mental and external actions enable two forms of extensibility. Also, the possibility to program the deliberation process in Java allows the programmer to define new language components.

Verification

- Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?
- A: The formal semantics of the language provides the basis for the formal verification of 3APL programs, both for theorem proving and model-checking approaches.

Software Engineering Principles

- Q:** Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?
- A:** Limited forms of abstraction, modularity and reusability are supported. Also, since the deliberation cycle and the shared environment are programmable as separate modules, the principle of separation of concern is respected.

Language Integration

- Q:** Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?
- A:** The 3APL platform and interpreter are programmed in Java. By means of external actions, Java can be called from the 3APL program. Further, a Java implementation of Prolog is used to implement the belief base of agents.

Language Integration (continued)

Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?

A: -

Platform

Deployment and Portability

- Q:** Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A:** There is a user guide that explains the use of the 3APL platform and the 3APL programming language through examples that are also available with the distribution. This user guide is under constant development.

Deployment and Portability (continued)

- Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?
- A: The 3APL platform can be run on Windows, Linux and Unix (Solaris) machines on which Java 2 SDK 1.4 is installed.

Standards Compliance

- Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?
- A: The 3APL platform supports limited naming and yellow page services.

Platform Extensibility

Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?

A: The 3APL platform is not open source yet, but the source is available on request.

Available Tools

- Q:** What tools are provided by the platform for the management, monitoring, logging and debugging of applications?
- A:** The platform provides a simple editor to write and modify individual agent programs. It also provide different execution modes such as single and multi-agent systems, either in a step-by-step or continuous fashion. Finally, it provides debugging tools such as different windows to observe the internal state of individual agents and a sniffer tool that visualizes the communication between agents.

Available Tools (continued)

- Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?
- A: The only available documentation is the user guide. An online tutorial will be available soon.

Available Tools (continued)

Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?

A: The 3APL platform provides a simple form of an IDE.

Tool Integration

Q: In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?

A: Since the 3APL platform is written in Java, any tools that can be integrated with Java can be integrated with the 3APL platform as well.

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** The applications run on the 3APL platform should be programmed in terms of individual agents that are programmed in the 3APL language.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: Applications with a maximum of 5 agents can be run efficiently within a single instance of the platform.

Performance Issues (continued)

Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?

A: The 3APL platform is an advanced prototype platform.

Multi-Agent Systems Features

- Q: Does the platform support open multi-agent systems and heterogeneous agents?
- A: The current platform does not support open and heterogeneous multi-agent systems.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: The platform provides only distributed control of agents.

Multi-Agent Systems Features (continued)

Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?

A: The 3APL platform does not offer libraries yet.

Applications Supported by the Language and Platform

Typical Examples

- Q:** What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?
- A:** Typical implemented examples are auctions, applications using Contract-Net protocols, cooperative systems, Axelrod's tournament and simple logistic applications.

Targeted Domains

Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?

A: Resource allocation, social simulation and all kinds of applications that can be described by BDI agents are target applications.

References I



R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge.

Model checking AgentSpeak.

In Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03), pages 409–416, Melbourne, 2003.



M. Dastani, F. de Boer, F. Dignum, and J.-J. Meyer.

Programming agent deliberation: An approach illustrated using the 3APL language.

In Proceedings of The Second Conference on Autonomous Agents and Multi-agent Systems (AAMAS'03), pages 97–104, Melbourne, 2003.



K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer.

Agent programming in 3APL.

Int. J. of Autonomous Agents and Multi-Agent Systems, 2(4):357–401, 1999.

References II



A. S. Rao and M. Georgeff.

BDI Agents: from theory to practice.

In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.



J. Thangarajah, L. Padgham, and M. Winikoff.

Detecting and avoiding interference between goals in intelligent agents.

In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 2003.



J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer.

Avoiding resource conflicts in intelligent agents.

In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence 2002 (ECAI 2002)*, Lyon, France, 2002.

References III



M. B. van Riemsdijk, F. S. de Boer, and J.-J. Ch Meyer.

Dynamic logic for plan revision in intelligent agents.

In Joao A. Leite and Paolo Torroni, editors, *Proceedings of the fifth international workshop on computational logic in multi-agent systems (CLIMA'04)*, pages 196–211, 2004.



M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer.

Agent programming in Dribble: from beliefs to goals using plans.

In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 393–400, Melbourne, 2003.

References IV



M. Birna van Riemsdijk, J.-J. Ch Meyer, and F. S. de Boer.

Semantics of plan revision in intelligent agents.

In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST04)*, volume 3116 of *LNCS*, pages 426–442. Springer-Verlag, 2004.



F. Zambonelli, N. R. Jennings, and M. Wooldridge.

Developing multiagent systems: The Gaia methodology.

ACM Transactions on Software Engineering and Methodology (TOSEM), 12(3):317–370, 2003.

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.

Abstract (continued)

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.

Authors

Amal El Fallah Seghrouchni and Alexandru Suna

- 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 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.
- 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.

- The mobile agents technology tries to improve the systems' performances since it provides powerful programming constructs for designing distributed and mobile applications.
- It becomes easy to design **active entities** that move over the network and perform tasks on hosts (**target sites** or **computers**).

- The main focus remains on the development of mobile objects and processes.
- The mobile agents are 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.

Programming environment - 3 main objectives

1. Propose an agent oriented programming language that:

- Helps to reduce the gap between design and implementation phases;
- Allows representation of cognitive skills (knowledge, beliefs, goals, 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;
- allows the dynamic adaptability and reconfiguring of the MAS. Agents are able to
 - Reconfigure themselves autonomously;
 - Acquire new knowledge and capabilities;
 - Dynamically adapt their structure (changes in environment and demands of target applications).

Programming environment - 3 main objectives I

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

System

- CLAIM (Computational Language for Autonomous, Intelligent and Mobile agents)
 - Intelligence;
 - Autonomy;
 - Communication primitives;
 - Cognitive skills.
- SyMPA (SYstem Multi-Platform of Agents)

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

Specifications and Syntactical Aspects

- A MAS in CLAIM is a set of hierarchies of agents distributed over a network.
- The notion of hierarchy can be seen as a membership relation:
 - “an agent is sub-agent of another agent” means that he is contained in the higher-level agent.

CLAIM agent

A CLAIM agent:

- Is a node in a hierarchy;
- Is an autonomous, intelligent and mobile entity that can be seen as a bounded place where the computation happens;
- has a parent, a list of local processes and a list of sub-agents;
- has intelligent components (knowledge, capabilities, goals) that allow a reactive or proactive behavior.

Agents and classes of agent

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,)* ) {...}
```

New agent

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

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

- Variables (denoted by $?x$) can be used to replace agents' names, messages, goals, etc.
- Two types of variables:
 - Global (for a class);
 - Local (to a capability).
- The agents' components allow representing the agents' mental state, communication and mobility.
- 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**.

Authority

- 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.

Parent

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.

Knowledge

The **knowledge** component contains information about other agents (about capabilities or classes) or about the world (divers propositions).

This knowledge base is a set of elements of *knowledge* type, defined as follows:

$$\begin{array}{lcl} \textit{knowledge} ::= & \textit{agentName}(\textit{capabilityName}, \textit{message}, \textit{effect}) \\ & | & \textit{agentName}:\textit{className} \\ & | & \textit{proposition} \end{array}$$

Note: The knowledge about other agents has a standard format, containing the name, his class or capability.

Knowledge (continued)

The user can define his own ontology of information about the world, represented as propositions containing a name and a list of arguments.

$proposition = name(arg_1, arg_2, \dots, arg_n)$

Propositions can also be used for denoting goals or messages.

Goals

- 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.

Messages

Every agent has a queue for storing the received **messages**.

- FIFO policy;
- A message from the queue contains the sender of the message and the arrived message:

queueMessage ::= agentName > message

Messages (continued)

An agent can send messages to an agent (**unicast**), (**multicast**), (**broadcast**), using the primitive: **send**(*receiver,message*), where the receiver can be:

this Himself;

parent The agent's current parent;

authority The agent's authority (the agent that created the current agent);

agentName The specified agent;

all All running agents;

?Ag:className All the agents that have been instantiated from the specified class of agents.

Messages (continued)

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.
 - ***tell(knowledge)***;
 - ***askAllCapabilities()***;
 - ***askIfCapability(capabilityName)***;
 - ***achieveCapability(capabilityName)***;
 - ***askEffect(effect)***;
 - ***doneEffect(effect)***.

Messages (continued)

In CLAIM there are three types of messages:

3. The **mobility messages** are used by the system during the mobility operations, for asking, granting or not granting mobility permissions:
 - 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;
 - The mobility messages is for an advanced security and control.

Capabilities

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;)+ }  
}
```

- 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.

- 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)+ )
```

Processes

An agent concurrently executes several **processes**:

- Sequence of processes;
- An instruction;
- A variable's instantiation;
- A method implemented in other programming language;
- 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.

Processes (Continued)

```
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)
```

Definition 3.1

- ***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.
- 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.
- 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?**".

Mobility argument

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:

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;
- Several capabilities can be concurrently activated.

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.

Semantics and Verification

Definition 3.2

- A MAS in CLAIM is a set of connected hierarchies of agents.
- At the semantical level, a MAS is a set Π of running agents.
- $\alpha, \beta, \pi, \dots$ are agents' names.
- a_1, a_2, \dots 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: $\rho = n(t_1, t_2, \dots, t_m)$.
- The other notations will be explained as they are introduced.

Definition 3.3 (Program)

A program is: $\Pi = a_1 \parallel a_2 \parallel \dots \parallel a_n, n \geq 0$.

The notation \parallel represents concurrent agents inside the MAS, running on the same computer or on different connected computers.

Definition 3.4 (Agent)

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

- α is the agent's name;
- π is the name of the agent's current parent;
- K is the knowledge base.

$$K = \{k_1, k_2, \dots, k_n\}, k_i = \rho_i \mid \alpha_i(n_i, m_i, E_i) \mid \alpha_i : cl_i$$
- G is the agent's set of current goals (not treated yet);
- 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:

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

Definition 3.5 (Agent (continued))

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

- C is the agent's list of capabilities. A capability
 - Has a name (n_i);
 - 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, a condition about the agent's knowledge, sub-agents or effects, or a logical formula.

We defined a function $V : (\Omega, \Pi) \rightarrow \{true, false\}$, that evaluates the boolean value of a CLAIM agent condition in the context of a running MAS.

Definition 3.6 (Agent (continued))

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

- P is the list of the agent's concurrent running processes
(the notation $|$ represents concurrent processes inside an agent):

$$P ::= p_i \mid p_j \mid \dots \mid p_k$$

- S is the set of names of the agent's sub-agents;
- E is the list of achieved goals or effects.

Definition 3.7 (Agent (continued))

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

Semantics and Verification

Additional notations

- Propositions are important notions in our language.
- A proposition has a name and contains a set of arguments:
 $\rho = n(t_1, t_2, \dots, t_m)$.
- 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 3.8

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

Definition 3.9

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

Definition 3.10

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

Definition 3.11

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.

Semantics and Verification

Conditions

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

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

Semantics and Verification

Reduction rules

Recap: A program contains a set of concurrent running agents:

$\Pi = a_1 \parallel a_2 \parallel \dots a_n$, where \parallel represents concurrent running agents in the MAS.

- For representing the semantics of CLAIM programs we choose an operational approach consisting in a transition relation \rightarrow between states of a program.
- 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 $\Pi \rightarrow \Pi'$).

- 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.

Semantics and Verification

Terminal configuration

Definition 3.12 (Program Termination)

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

Definition 3.13 (Agent Termination)

A CLAIM agent is in a **terminal configuration** if he has no message or goal to treat and no running process.

Example 3.14

$$a_i = \langle \alpha, \pi, K, \emptyset, \emptyset, \emptyset, C, \emptyset, S, E \rangle$$

Even if an agent can still receive messages that activate capabilities, we call this kind of configuration a terminal configuration.

Semantics and Verification

Message transmission

Using the *send* primitive and the language's possibilities, an agent can send a message to:

- Himself;
- Another agent;
- All the agents belonging to a class (multicast);
- All the agents in the MAS.

The message is added at the end of the messages queue M :

Rule ?? (send)

$$\langle \alpha, send(\beta, m) \rangle \parallel \langle \beta, M \rangle \rightarrow \langle \alpha, \emptyset \rangle \parallel \langle \beta, M.\alpha\{m\} \rangle \quad (1)$$

Semantics and Verification

Message processing

- The **arrived messages are processed sequentially**, following a FIFO policy.
- **Own messages** or several **pre-defined messages** (e.g. *tell*, *askIfCapability*, *askAllCapabilities*, *achieveCapability*, *askEffect*, *doneEffect*) for exchanging information about capabilities, effects and knowledge base.
- These messages have a pre-defined treatment.

- 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 somehow else (*e.g.* verifying the trust level of the sender).

Rule ?? (tell)

The *tell* message, used by an agent to send a piece of information to another agent:

$$\langle \beta, K, \alpha\{tell(k)\} \rangle \rightarrow \langle \beta, K \cup \{k\}, \emptyset \rangle \quad (2)$$

- If the triggering message of a capability arrives and its condition is verified, the associated processes are executed and the effects are updated (rule ??).
 - When a message arrives, the variables in the condition, process or effects are replaced with the corresponding values sent in the message.
- 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.

Rule ?? (cap)

$$\frac{\langle \beta, \alpha\{m\}, C, \emptyset \rangle, \text{ and } \exists \langle n_i, m_i, \Omega_i, p_i, E_i \rangle \in C, m_i \cong m \text{ and } V(\Omega'_i) = \text{true}}{\langle \beta, \emptyset, C, p'_i.\text{addEffect}(e'_1)....\text{addEffect}(e'_j) \rangle, e'_1...e'_j \in E'_i} \quad (3)$$

- 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.

Semantics and Verification

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 ??).

Rule ?? (cond)

$$\frac{\langle \beta, C, \emptyset \rangle, \text{ and } \exists \langle n_i, \emptyset, \Omega_i, p_i, E_i \rangle \in C, V(\Omega_i) = \text{true}}{\langle \beta, C, p_i.\text{addEffect}(e_1)....\text{addEffect}(e_j) \rangle, e_1...e_j \in E_i} \quad (4)$$

Semantics and Verification

Agents' creation and removal

When an agent is created using the *newAgent* operation, his components are instantiated from an already defined class:

Rule ?? (new)

$$\frac{\langle \alpha, \text{newAgent} \langle \beta, \emptyset, K, G, \emptyset, \emptyset, C, P, \emptyset, \emptyset \rangle, \emptyset \rangle}{\langle \alpha, \emptyset, \{\beta\} \rangle \parallel \langle \beta, \alpha, K, G, \emptyset, \emptyset, C, P, \emptyset, \emptyset \rangle} \quad (5)$$

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

Rule ?? (kill)

$$\frac{\langle \alpha, \pi, \text{kill}(\beta), S_\alpha \rangle \parallel \langle \beta, \alpha \rangle, \text{ where } \beta \in S_\alpha}{\langle \alpha, \pi, \emptyset, S_\alpha - \{\beta\} \rangle} \quad (6)$$

Semantics and Verification

Mobility operations

- The **mobility primitives** are inspired from the **ambient calculus**.
- Using *in*, an agent can enter another agent from the same level in the hierarchy (rule ?? and Figure ??)
- Using *out*, an agent can exit his parent (rule ?? and Figure ??).
- Unlike the **ambient calculus**, where there is no control, we added an asking/granting permission mechanism, represented in term of **co-actions**.
- By default, a CLAIM agent will receive these permissions, unless another agent is explicitly programmed to refuse to give them.

Rule ?? (in)

$$\frac{\langle \pi, S_\pi \rangle \parallel \langle \alpha, \pi, in(\beta) \rangle \parallel \langle \beta, \pi, \overline{in}(\alpha), S_\beta \rangle, \alpha, \beta \in S_\pi}{\langle \pi, S_\pi - \{\alpha\} \rangle \parallel \langle \alpha, \beta, \emptyset \rangle \parallel \langle \beta, \pi, \emptyset, S_\beta \cup \{\alpha\} \rangle} \quad (7)$$

Rule ?? (out)

$$\frac{\langle \pi, S_\pi \rangle \parallel \langle \alpha, \beta, out(\beta) \rangle \parallel \langle \beta, \pi, \overline{out}(\alpha), S_\beta \rangle, \beta \in S_\pi, \alpha \in S_\beta}{\langle \pi, S_\pi \cup \{\alpha\} \rangle \parallel \langle \alpha, \pi, \emptyset \rangle \parallel \langle \beta, \pi, \emptyset, S_\beta - \{\alpha\} \rangle} \quad (8)$$

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.

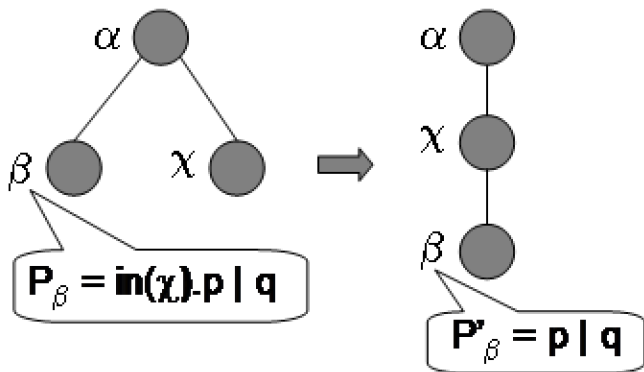


Figure 8: The enter operation

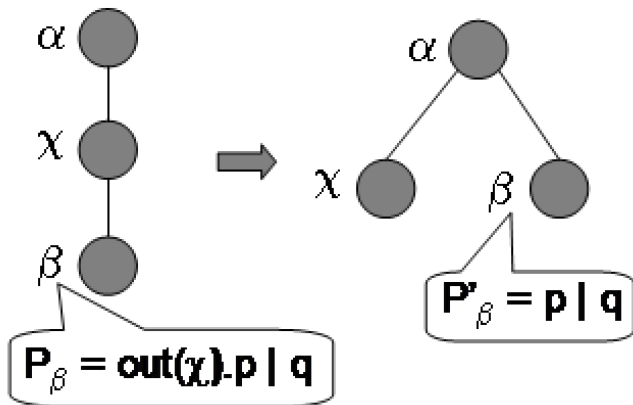


Figure 9: The exit operation

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

Rule ?? (move)

$$\frac{\langle \pi, \overline{out}(\alpha), S_\pi \rangle \parallel \langle \alpha, \pi, move(\beta), S_\alpha \rangle \parallel \langle \beta, \overline{in}(\alpha), S_\beta \rangle, \alpha \in S_\pi}{\langle \pi, \emptyset, S_\pi - \{\alpha\} \rangle \parallel \langle \alpha, \beta, \emptyset, S_\alpha \rangle \parallel \langle \beta, \emptyset, S_\beta \cup \{\alpha\} \rangle} \quad (9)$$

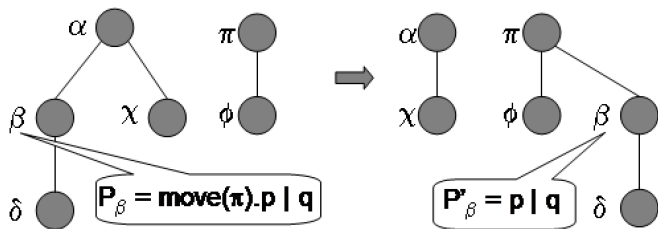


Figure 10: The move operation

- The *open* and *acid* actions are used:
 - For opening one of the sub-agents (rule ?? and Figure ??);
 - For opening his own boundaries (rule ?? and Figure ??).
- 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.
- 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.

Rule ?? (open)

$$\frac{\langle \alpha, K_\alpha, C_\alpha, P \mid \text{open}(\beta), S_\alpha \rangle \parallel \langle \beta, \alpha, K_\beta, C_\beta, Q \mid \overline{\text{open}}(\alpha), S_\beta \rangle \parallel a_\beta \quad \text{where } \beta \in S_\alpha, a_\beta = \langle \gamma_\beta, \beta \rangle, \forall \gamma_\beta \in S_\beta}{\langle \alpha, K_\alpha \cup K_\beta, C_\alpha \cup C_\beta, P \mid Q, S_\alpha \cup S_\beta \rangle \parallel a_\beta, \text{ where } a_\beta = \langle \gamma_\beta, \alpha \rangle, \forall \gamma_\beta \in S_\beta} \quad (10)$$

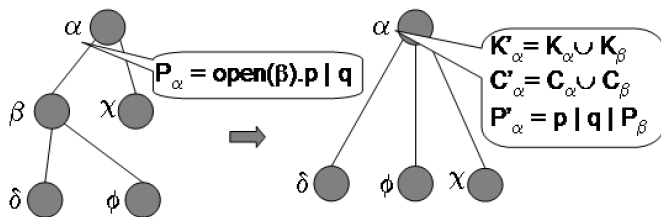


Figure 11: The open operation

Rule ?? (acid)

$$\frac{\langle \alpha, K_\alpha, C_\alpha, P \mid \overline{acid}(\beta), S_\alpha \rangle \parallel \langle \beta, \alpha, K_\beta, C_\beta, Q \mid acid, S_\beta \rangle \parallel a_\beta \quad \text{where } \beta \in S_\alpha, a_\beta = \langle \gamma_\beta, \beta \rangle, \forall \gamma_\beta \in S_\beta}{\langle \alpha, K_\alpha \cup K_\beta, C_\alpha \cup C_\beta, P \mid Q, S_\alpha \cup S_\beta \rangle \parallel a_\beta, \text{ where } a_\beta = \langle \gamma_\beta, \alpha \rangle, \forall \gamma_\beta \in S_\beta} \quad (11)$$

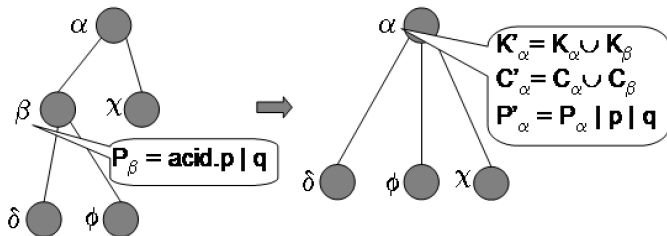


Figure 12: The acid operation

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

Semantics and Verification

Instructions

There are two instructions in CLAIM:

- 1 *forAllKnowledge* allows to sequentially execute a process for all the elements in the knowledge base verifying a criterion (rule ??).
- 2 *forAllAgents* allows to execute a process for all the sub-agents verifying a certain criterion (e.g. all sub-agents - rule ??, or all sub-agents belonging to a specific class - rule ??).

Rule ?? (forAllKnowledge)

$$\frac{\langle \alpha, K, \text{forAllKnowledge}(k)\{p_i\} \rangle}{\langle \alpha, K, p_i\{k_1/k\} \dots p_i\{k_j/k\} \rangle, \forall k_i \in K, 1 \leq i \leq j, k \cong k_i} \quad (12)$$

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 .

Rule ?? (forAllAgents)

$$\frac{\langle \alpha, K, \text{forAllAgents}(\text{?}x)\{p_i\}, S \rangle}{\langle \alpha, K, p_i\{\gamma_1/\gamma\} \dots p_i\{\gamma_n/\gamma\}, S \rangle, \forall \gamma_i \in S, 1 \leq i \leq n} \quad (13)$$

Rule ?? (forAllAgents belongs to the class)

$$\frac{\langle \alpha, K, \text{forAllAgents}(\text{?}x : cl)\{p_i\}, S \rangle, S_{cl} \subseteq S, \forall \gamma_i \in S_{cl}, \gamma_i : cl, \gamma_i \text{ bc } cl}{\langle \alpha, K, p_i\{\gamma_1/\gamma\} \dots p_i\{\gamma_j/\gamma\}, S \rangle, \forall \gamma_i \in S_{cl}, 1 \leq i \leq j} \quad (14)$$

bc : belongs to the class.

Semantics and Verification

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.

Rule ?? (effect)

$$\frac{\langle \alpha, G, G', \text{addEffect}(e_i), E \rangle}{\langle \alpha, G - \{e_i\}, G' - \{e_i\}, \emptyset, E \cup \{e_i\} \rangle} \quad (15)$$

Semantics and Verification

The goal-driven behavior

- Beside the **reactive behavior** 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 (p'_i, Ω'_i and $e'_1 \dots e'_j$ have the variables replaced with values from g):

Rule ?? (goal1)

$$\frac{\langle \alpha, \{g\}, \emptyset, C, \emptyset \rangle, \exists \langle s_i, m_i, \Omega_i, p_i, E_i \rangle \in C, \exists e_i \in E_i, e_i \cong g, V(\Omega'_i) = true}{\langle \alpha, \emptyset, \{g\}, C, p'_i.addEffect(e'_1) \dots addEffect(e'_j) \rangle, e'_1 \dots e'_j \in E_i} \quad (16)$$

- 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 first goal is moved from the current goals list to the processing goals:

Rule ?? (goal2)

$$\frac{\langle \alpha, \{g\}, \emptyset, C \rangle, \exists \langle s_i, m_i, \Omega_i, p_i, E_i \rangle \in C, \exists e_i \in E_i, e_i \cong g, \\ V(\Omega_i) = \text{false}, \Omega_i \text{ contains } \textit{this.e}_i}{\langle \alpha, \{e'_i\}, \{g\}, C \rangle} \quad (17)$$

- 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 ?? (askEffect)

$$\frac{\langle \alpha, \{g\}, \emptyset, C, \emptyset \rangle, \exists \langle s_i, m_i, \Omega_i, p_i, E_i \rangle \in C, \exists e_i \in E_i, e_i \cong g, \\ V(\Omega_i) = \text{false}, \Omega_i \text{ contains } \beta.e_i}{\langle \alpha, \emptyset, \{g\}, C, \text{send}(\beta, \text{askEffect}(e_i')) \rangle} \quad (18)$$

- When an agent receives an *askEffect* message, if he does not have a capability with this message, he will add the demanded effect to his list of goals:

Rule ?? (message askEffect)

$$\langle \beta, \emptyset, \alpha \{askEffect(e_i)\} \rangle \rightarrow \langle \beta, \{\alpha.e_i\}, \emptyset \rangle \quad (19)$$

- 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 ?? (doneEffect)

$$\frac{\langle \beta, G, G', addEffect(e_i), E \rangle, \text{ and } \exists \alpha. e_i \in G \text{ or } \exists \alpha. e_i \in G'}{\langle \beta, G - \{e_i\}, G' - \{e_i\}, send(\alpha, doneEffect(e_i)), E \cup \{e_i\} \rangle} \quad (20)$$

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.

Semantics and Verification

Variable instantiation

The language allows to **instantiate variables** that will be used in the following processes in the current sequence:

Rule ?? (variable)

$$\langle \alpha, x = v.p_i \rangle \quad \rightarrow \quad \langle \alpha, p_i\{v/x\} \rangle \quad (21)$$

Semantics and Verification

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 :

Rule ?? (sequence)

$$\text{if } \langle \alpha, p_i \rangle \rightarrow \langle \alpha, p'_i \rangle \text{ then } \langle \alpha, p_i.q \rangle \rightarrow \langle \alpha, p'_i.q \rangle \quad (22)$$

Semantics and Verification

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.

Semantics and Verification

Verification of programs: A discussion

- The operational semantics 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 and verification.
- A CLAIM program is distributed and concurrent, containing agent communicating asynchronously and that do not share common variables.

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:

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

This configuration can evolve (with equal probabilities) in two different configurations.

If α executes **in**:

$$\begin{aligned} \langle \tau, S_\tau \cup \{\pi\} \rangle & \parallel \langle \pi, \overline{out}(\beta).p_k, S_\pi \cup \{\beta\} \rangle \parallel \\ \langle \alpha, \beta, p_i \rangle & \parallel \langle \beta, \pi, out(\pi).p_l \mid p_j, S_\beta \cup \{\alpha\} \rangle \end{aligned}$$

or, if **out** is executed by β :

$$\begin{aligned} \langle \tau, S_\tau \cup \{\pi, \beta\} \rangle & \parallel \langle \pi, p_k, S_\pi \cup \{\alpha\} \rangle \parallel \\ \langle \alpha, \pi, in(\beta).p_j \rangle & \parallel \langle \beta, \tau, p_l \mid \overline{in}(\alpha).p_j, S_\beta \rangle \end{aligned}$$

- In the first case, β will still be capable of executing $out(\pi)$;
- In the second case, α no longer can enter β , because he is not at the same level in the hierarchy anymore.
- 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 an agent may try infinitely to execute an *in* operation, for entering an agent that is not in his neighborhood, and consequently the next processes 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.
- 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*.
 - The equivalence between programs is reduced at equivalence between processes inside agents.

Structural congruence (continued)

- Processes are grouped into equivalence classes using the structural congruence relation \equiv .

$$p \equiv p$$

$$p \equiv q \Rightarrow q \equiv p$$

$$p \equiv q, q \equiv r \Rightarrow p \equiv r$$

$$p \mid 0 \equiv p$$

$$p.0 \equiv p$$

$$p \mid q \equiv q \mid p$$

$$(p \mid q) \mid r \equiv p \mid (q \mid r)$$

$$p \equiv q \Rightarrow p \mid r \equiv q \mid r$$

$$p \equiv q \Rightarrow p.r \equiv q.r$$

$$p \equiv q \Rightarrow r.p \equiv r.q$$

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.

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 weak 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.

Available tools and documentation

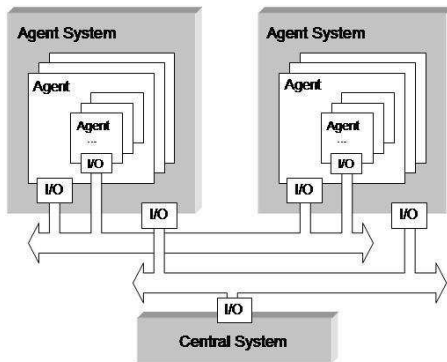


Figure 13: SyMPA's Architecture

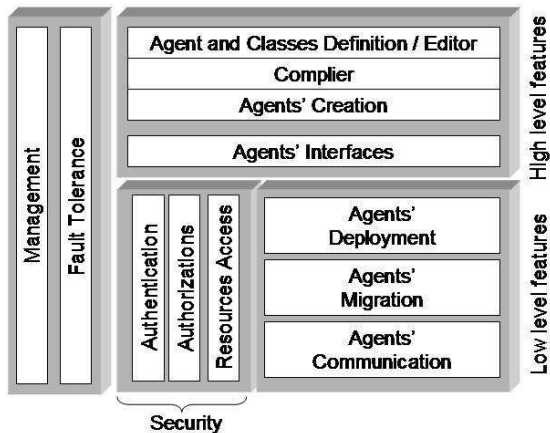


Figure 14: SyMPA's features

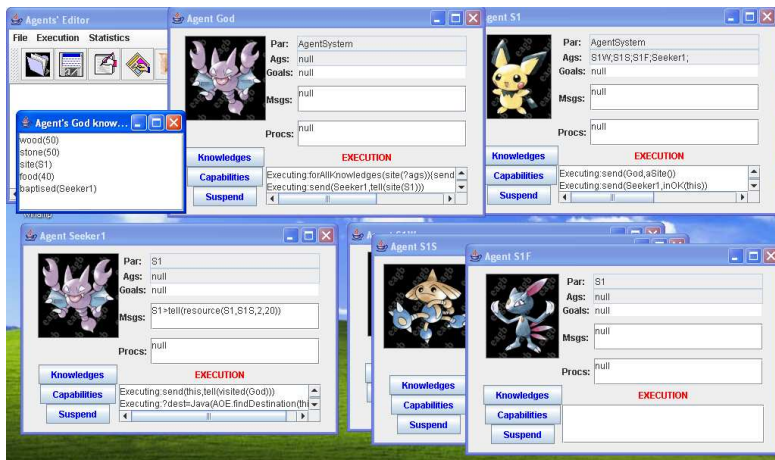


Figure 15: Agents' interfaces

Available tools and documentation

Mobility

- The **local migration** takes place inside a hierarchy.
- The **remote migration** is the migration between hierarchies, using the *move* primitive.

The **remote mobility** in SyMPA can be considered at two levels:

1

- There is a strong migration at the language level.
- The agent's 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.
- This representation is sent through the network to the destination agent system.

2

- 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.

Available tools and documentation

Security

The mobile agents are programs running in a distributed and **insecure environment** (e.g. the Internet). For the agent systems' protection, we are using:

- **Agents' authentication**;
- **Encryption** during the migration and during the execution on a agent system (when the agent is stored on the disk);
- **Fault-tolerance mechanisms**.

Standards compliance, interoperability and portability

- The **SyMPA platform** is implemented in Java.
- The **SyMPA environment** is an ensemble of packages for the *Java Virtual Machine*.
- A few configuring operations are needed and the CLAIM language supported by the platform is ready to be used to implement MAS applications.
- Installed and tested the platform on Windows, Unix-based or Macintosh systems.
- Compliant with the specifications of the **MASIF standard** from the OMG.

Other features of the platform

- Concerning the reached performances, we could deploy up to 30 agents on one computer.
- In our current applications we used the interfaces to monitor the agents' execution, behavior, communication and migration.
- Until now we developed application using agents deployed on up to 10 connected computers.
- 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.
- 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 is 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.

Translations

In the first phase of development, applications from other agent-oriented programming languages, such as:

- **Airline reservations** from **AGENT-0**;
- **Bolts Make Scenario** from **AgentSpeak**;

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 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, 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

- 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.
- 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.
- 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.
- 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**:

- 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.

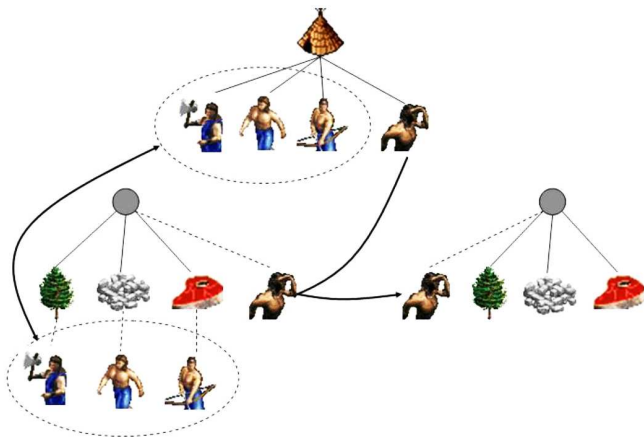


Figure 16: Application's schema

The **Creator** agent:

- Creates (using **newAgent**) a **Seeker** agent;
- Finds out the list of the existing sites;
- Tells to the **Seeker** to migrate to each of them (using **move**).

When the **Seeker** arrives on a site:

- He “counts” the available resources;
- He 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.

```
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;
}
```

```

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;
}

```

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** The CLAIM language is suitable to design stationary or mobile intelligent agents, having a powerful mental state containing knowledge, goals and capabilities, allowing an autonomous, reactive or goal-oriented behavior.

Communication

- Q:** Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A:** CLAIM supports unicast, multicast or broadcast communication between agents. It offers a set of predefined messages inspired from the speech-acts theory but also leaves the possibility to the designer to define his own messages.

Underlying Computational Model

- Q:** Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?
- A:** The CLAIM language offers support for the agents' migration as a main feature. The CLAIM agents are both intelligent and mobile. There is a strong mobility at the agents' processes level and a weak mobility for the invoked Java methods.

Simplicity

Q: How easy it is to use and understand the language?

A: The facility in developing several simple or complex applications proved that CLAIM is easy to use.

Preciseness

Q: Does the language have clear and precise semantics? How has it been formalised?

A: CLAIM has a formal operational semantics consisting in a set of reduction rules between states of programs.

Expressiveness

- Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?
- A: The variety of implemented applications proved the expressiveness of CLAIM.

Extensiveness

- Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?
- A: The language offers to the developer the possibility to define his own ontology for agents' knowledge and his own messages and goals for agents.

Verification

Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?

A: The language's operational semantics is a first important step towards the programs' verifications. Our current work tackles this aspect.

Software Engineering Principles

Q: Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?

A: The notion of generic class is central in CLAIM.

Language Integration

- Q:** Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?
- A:** The agents in CLAIM can invoke Java methods or Web Services for computational purposes. We intend to give the agents the possibility to invoke methods implemented in other programming languages.

Language Integration (continued)

Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?

A: -

Platform

Deployment and Portability

- Q:** Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A:** The platform contains installation and deployment guidelines. The documentation is represented by several published articles, concerning the language as well as the platform. A tutorial and a documentation of the API will be soon available.

Deployment and Portability (continued)

- Q:** Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?
- A:** The platform is implemented in Java, is portable and can be installed on every computer supporting *Java Virtual Machine*. So the platform is platform-independent. We have already tested it on Windows, Unix-based and Macintosh systems.

Standards Compliance

- Q:** To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?
- A:** SyMPA is compliant with the specifications of the MASIF standard from the OMG.

Platform Extensibility

Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?

A: The platform is extensible.

Available Tools

- Q:** What tools are provided by the platform for the management, monitoring, logging and debugging of applications?
- A:** The platform offers management and monitoring functions at the central system and at the agent system level. For every running agent there is a graphical interface for visualizing the agent's behavior, communication and migration. Momentary, there are no debugging tools.

Available Tools (continued)

Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?

A: -

Available Tools (continued)

- Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?
- A: Each agent system offers an editor for defining agents and classes of agents, a compiler for verifying the definitions' syntax and an execution engine for deploying agents.

Tool Integration

- Q:** In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?
- A:** The CLAIM agents can invoke Web Services. There is also an extension of SyMPA allowing to heterogeneous agents to interact using the Web Services features.

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** There are no specific operating system requirements. Being implemented in Java, the platform only requires the **JRE**.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: Until now, we performed tests with up to 30 communicating and mobile agents on a computer (including their graphical interfaces) and we deployed the platform on 10 connected computers. We intend to test the platform on a larger-scale environment.

Performance Issues (continued)

- Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?
- A: The platform is a prototype that served for developing several complex applications by different people.

Multi-Agent Systems Features

- Q:** Does the platform support open multi-agent systems and heterogeneous agents?
- A:** The platform supports open multi-agent systems. Agents are dynamically created and removed. Without any add-on, the platform supports only CLAIM agents. Nevertheless, we developed an interoperability environment that allows to heterogeneous agents to interact using a Web Services based approach.

Multi-Agent Systems Features (continued)

- Q:** Does the platform provide centralised or distributed control, and hierarchical structure of agents?
- A:** The agents in CLAIM are hierarchically represented. An agent has a parent and can have several sub-agents. In this version there is a centralized management but different management solutions will be available in the future.

Multi-Agent Systems Features (continued)

Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?

A: Classes of agents can be defined in CLAIM that can be parameterized and reused later.

Applications Supported by the Language and Platform

Typical Examples

- Q:** What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?
- A:** Our agent-oriented environment has already been used to develop several applications. One can easily design applications focused on the reasoning abilities of an agent, but the main purpose of CLAIM is to develop distributed applications that takes advantage of the agents' mobility and adaptability allowed by the language's features. The most prominent applications were those of electronic commerce and distributed libraries.

Targeted Domains

Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?

A: CLAIM can be used to develop a wide area of agent-based applications.

References I



K.A. Apt and Olderog E.-R.

Verification of Sequential and Concurrent Programs.

Springer Verlag, 1991.



L. Cardelli and A.D. Gordon.

Mobile ambients.

Foundations of Software Science and Computational Structures, LNAI,

1378:140–155, 1998.



A. El Fallah Seghrouchni, S. Haddad, T. Melitti, and A. Suna.

Interopérabilité des systèmes multi-agents à l'aide des services web (in french).

In *Proceedings of JFSMA'04*, Paris, 2004.

References II



A. El Fallah Seghrouchni and A. Suna.

Claim: A computational language for autonomous, intelligent and mobile agents.

Proceedings of ProMAS'03 Workshop of AAMAS, LNAI, 3067:90–110, 2003.



Oliver Enseling.

Build your own languages with JavaCC.

World Wide Web page.



M.S. Greenberg, J.C. Buyington, and D.G. Harper.

Mobile agents and security.

IEEE Communications Magazine, 36(7):76–85, 1998.



G. Klein, A. Suna, and A. El Fallah Seghrouchni.

A methodology for building mobile multi-agent systems.

In *Proceedings of SYNACS'04*, Timisoara, Romania, 2004.

References III



F. Levi and D. Sangiori.

Controlling interference in ambients.

In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 352–364. 2000.



D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White.

MASIF, the OMG mobile agent system interoperability facility.

In Proceedings of Mobile Agents, pages 50–67. 1998.



A. Suna and A. El Fallah Seghrouchni.

A mobile agents platform: architecture, mobility and security elements.

In Proceedings of ProMAS'04 Workshop of AAMAS, New-York, 2004.

References IV



A. Suna, C. Lemaitre, and A. El Fallah Seghrouchni.

E-commerce using an agent oriented approach.

In *Proceedings of the Iberagents Workshop*, Puebla, Mexico, 2004.

Abstract

The *IMPACT* project (<http://www.cs.umd.edu/projects/impact>) aims at developing a powerful multi-agent system platform, which (1) is able to deal with **heterogenous and distributed data**, (2) can be realized on top of **arbitrary legacy code**, (3) is built on a **clear foundational basis**, and (4) **scales up for realistic applications**. We will describe its main features and several extensions of the language that have been investigated (and partially implemented).

Keywords

formal methods, heterogeneity, legacy code, annotated logic programming, reasoning with time, uncertainty and beliefs

Authors

Jürgen Dix, Yingqian Zhang

Main features of *IMPACT*

- Idea of **agentisation**.
- **Clear semantics** for agents.
- Identify **classes of programs** that can be **efficiently implemented**.

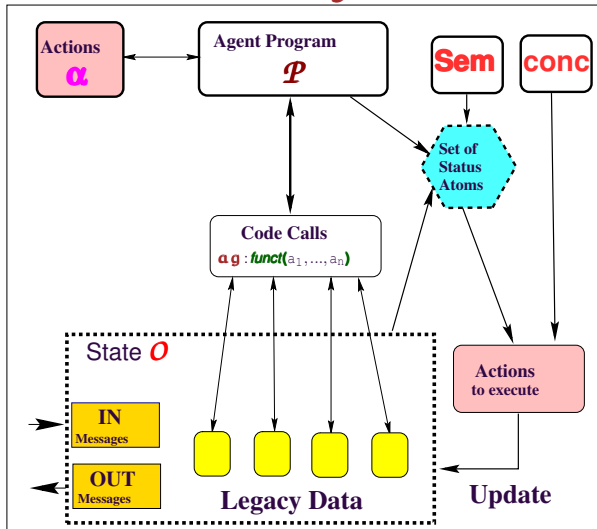
General remarks

Body of software code

$$\mathcal{S}^a =_{def} (\mathcal{T}_{\mathcal{S}^a}, \mathcal{F}_{\mathcal{S}^a}, \mathcal{C}_{\mathcal{S}^a}).$$

- $\mathcal{F}_{\mathcal{S}^a}$ Set of predefined functions which make access to the data objects.
- $\mathcal{T}_{\mathcal{S}^a}$ Data objects managed by the agent available to external processes.
- $\mathcal{C}_{\mathcal{S}^a}$ Composition operators to build new datatypes from the given ones.
- a Agent.

A Single agent



Agent properties

- Each *IMPACT* agent has certain **actions** α available.
- Agent act in their environment according to their **agent program** \mathcal{P}
- A well defined **semantics** Sem determining which of the actions the agent should execute.
- *IMPACT* agents are built on top of arbitrary software code
 $\mathcal{S}^a =_{\text{def}} (\mathcal{T}_{\mathcal{S}^a}, \mathcal{F}_{\mathcal{S}^a}, \mathcal{C}_{\mathcal{S}^a})$ (**Legacy Data**).
- A methodology for transforming arbitrary software (legacy code) into an **agent** has been developed.
- And a special **agent cycle**.

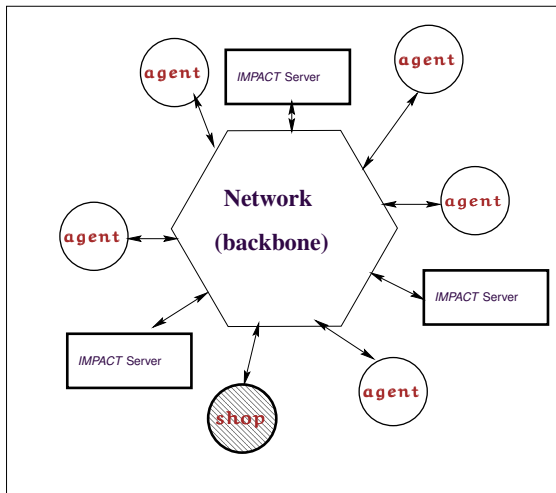
Each agent continually undergoes the following cycle:

- ① **Get messages** sent by other agents. This **changes the state O** of the agent.
- ② **Determine** (based on its program \mathcal{P} , its semantics **Sem** and its state O) **for each action α its status** (permitted, obliged, forbidden, ...). The agent ends up with a **set of status atoms**.
- ③ Based on a notion of concurrency **conc**, **determine the actions that can be executed** and update the state accordingly.

General architecture

- ① Agents communicate with other agents through the network (send/receive messages, ask server to find out about services that other agents offer).
- ② In many applications a **statistics** agent is needed (It keeps track of distances between two given points and the authorized range or capacity of certain vehicles)
 \rightsquigarrow This information can be stored in several databases.
- ③ Another example is the **supplier** agent. It determines through its databases which vehicles are accessible at a given location.

IMPACT Architecture



Definition 4.1 (State of an Agent, $O_{S(t)}$)

At any given point t in time, the **state of an agent**, denoted $O_{S(t)}$, is the set of all data objects that are currently stored in the relations the agent handles—the types of these objects must be in the base set of types in \mathcal{T}_S .

Example 4.2 (State)

The state of the **statistics** agent consists of all tuples stored in the databases it handles. The state of the **supplier** agent is the set of all tuples describing which vehicles are accessible at a given location.

Message box

There is a special datastructure, the **message box**, part of each agent. This message box is just one of those types. Thus a state change already occurs when a message is received.

Example 4.3 (Rescue Scenario I, temporal reasoning)

Consider a simplistic rescue operation where a natural calamity (e.g., a flood) has stranded many people. Rescuing these people requires close coordination between helicopters and ground vehicles. For the sake of this example, we assume the existence of:

- 1 A **heli**copter agent that conducts aerial reconnaissance and supports aerial rescues;
- 2 A set **gv1, gv2, gv3** of ground vehicles that move along the ground to appropriate locations—such vehicles may include ambulances as well as earth moving vehicles.
- 3 An immobile command centre agent **comc** that coordinates between the helicopter and the ground vehicles.

Example 4.4 (Rescue Scenario I, temporal reasoning (continued))

Here is a typical statement that should be expressible in an agent language.

If the maximal time previously taken to ship some equipment E from location A to location B is T_1 , and if equipment E is required to be at location B at time T , then ship E sometime between time $T - T_1 - 10$ and $T - T_1$.

Example 4.5 (Rescue Scenario II, agentising a planner)

- The planner SHOP is a stand-alone system.
- It uses **hierarchical task networks**.
- It requires that all **data is stored locally** and given in a particular format (atomic facts in Lisp notation).
- Such planning systems usually support only one kind of reasoning: **symbolic** or **numeric**, but not both.

Question

How can such a planning system be agentised in *IMPACT* as a planning agent A-SHOP?

◀ Back

Example 4.6 (Rescue Scenario II, agentising a planner (continued))

- Typical test domain: **Transportation planning problem for a rescue mission.**
- Computing plans involves performing a rescue mission where a task force is grouped and transported between an **initial location** (the assembly point) and the **NEO** site (where the evacuees are located).
- After the troops arrived at the *NEO* site, evacuees are re-located to a **safe haven.**

Example 4.7 (Rescue Scenario II, agentising a planner (continued))

The planning task involves:

- 1 **selecting** possible pre-defined routes, consisting of four or more segments each;
- 2 **choosing** a transportation mode for each segment;
- 3 **determining conditions** such as whether communication exists with *State Department personnel* and the type of evacuee registration process.

Example 4.8 (Rescue Scenario II, agentising a planner (continued))

Here we have four different *IMPACT* information sources available:

- **Transport Authority:** Maintains information about the transportation assets available at different locations.
- **Weather Authority:** Maintains information about the weather conditions at the different locations.
- **Airport Authority:** Maintains information about availability and conditions of airports at different locations.
- **Math Agent: *math*** evaluates arithmetic expressions. Typical evaluations include to subtract a certain number of assets use for an operation and update time delays.

⇒ **Agentising** given legacy code cannot be done automatically.

Specifications and Syntactical Aspects

Definition 4.9 (Software Code)

We may characterise the code on top of which an agent \mathbf{a} is built as a triple

$\mathcal{S}^{\mathbf{a}} =_{\text{def}} (\mathcal{T}_{\mathcal{S}^{\mathbf{a}}}, \mathcal{F}_{\mathcal{S}^{\mathbf{a}}}, \mathcal{C}_{\mathcal{S}^{\mathbf{a}}})$ where:

- 1 $\mathcal{T}_{\mathcal{S}^{\mathbf{a}}}$ is the set of all data types managed by \mathcal{S} ,
- 2 $\mathcal{F}_{\mathcal{S}^{\mathbf{a}}}$ is the set of predefined (API) functions over $\mathcal{T}_{\mathcal{S}^{\mathbf{a}}}$ through which external processes may access \mathbf{a} 's data
- 3 $\mathcal{C}_{\mathcal{S}^{\mathbf{a}}}$ is a set of type composition operations. A type composition operator is a partial n -ary function c which takes as input types τ_1, \dots, τ_n and yields as output a type $c(\tau_1, \dots, \tau_n)$.

Each agent also has a **message box** having a well defined set of associated code calls that can be invoked by external programs.

Example 4.10 (Rescue Scenario I)

Consider the rescue mission described earlier. The **heli** agent may have the following data types and code calls.

- Data Types: *speed*, *bearing* of type `int`, *location* of type `point` (record containing *x*, *y*, *z* fields), *nextdest* of type `string`, and *inventory*—a relation having schema $(\text{Item}, \text{Qty}, \text{Unit})$.
- Functions:
 - **heli**: *location*() : which returns the (x, y, z) coordinates of the current position of the helicopter.
 - **heli**: *inventory*(*Item*) : returns a pair of the form $\langle \text{Qty}, \text{Unit} \rangle$. For example, **heli**: *inventory*(*blood*) may return $\langle 25, \text{litres} \rangle$ specifying that the helicopter currently has 25 units of blood available.

An agent's state change because it took an action or because it received a message. another agent **b** cannot directly change **a**'s state.

Example 4.11 (Rescue Scenario I: State)

For instance, at a given instant of time, the state of the **heli** agent may consist of *location* = $\langle 45, 50, 9000 \rangle$, and *inventory* containing the tuples:
 $\langle \text{fuel}, 125, \text{gallons} \rangle$, $\langle \text{blood}, 25, \text{litres} \rangle$, $\langle \text{bandages}, 50, - \rangle$, $\langle \text{cotton}, 20, \text{lbs} \rangle$.

Queries and/or conditions may be evaluated w.r.t. an agent state using the notion of a code call atom and a code call condition (**CCC**) defined below.

Definition 4.12 (Code Call (CC**)/Code Call Atom)**

If **S** is the name of a software package, f is a function defined in this package, and (d_1, \dots, d_n) is a tuple of arguments of the input type of f , then the term **S**: $f(d_1, \dots, d_n)$ is called a **code call** (denoted by **CC**).

If cc is a code call, and X is either a variable symbol, or an object of the output type of cc , then **in**(X, cc) is called a **code call atom**.

If X is a variable over type τ and τ is a record structure with field f , then $X.f$ is a variable ranging over objects of the type of field f .

Definition 4.13 (Code Call Condition (CCC))

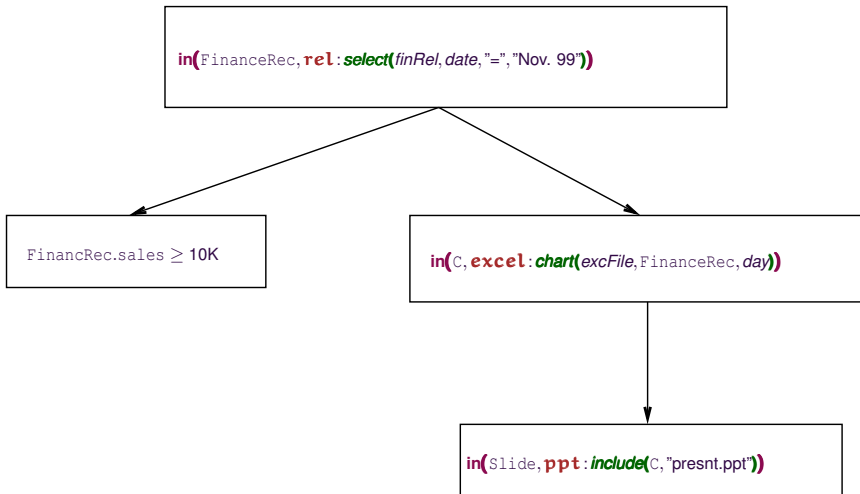
- ① Every code call atom is a **code call condition**.
- ② If s, t are either variables or objects, then $s = t$ is a **code call condition**.
- ③ If s, t are either integers/real valued objects, or are variables over the integers/reals, then $s < t, s > t, s \geq t, s \leq t$ are **code call conditions**.
- ④ If χ_1, χ_2 are **code call conditions**, then $\chi_1 \& \chi_2$ is a **code call condition**.

Example 4.14

`in(X, heli:inventory(fuel)) & X.Qty < 50` is a code call condition that is satisfied whenever the helicopter has less than 50 gallons of fuel left. The code call condition

```
in(FinanceRec, rel:select(finRel, date, "=", "Nov. 99")) &  
FinanceRec.sales ≥ 10K &  
in(C, excel:chart(excFile, FinanceRec, day)) &  
in(Slide, ppt:include(C, "presnt.ppt"))
```

is a complex condition that accesses and merges data across a relational database, an Excel file, and a PowerPoint file.



In the above example, it is very important that the first code call be evaluable. If, for example, the constant *finRel* were a variable, then

```
rel:select(finRel,date,"=", "Nov. 99")
```

would not be evaluable, unless there were another condition instantiating this variable.

Actions in *IMPACT*

An Action is a partial function from states to states, implemented by a **body of code**

The Agent reasons about actions via a set of **preconditions** and **effects** defining the conditions an agent state must satisfy for the action and the new state that results from such an execution.

We assume that the preconditions and effects associated with an action correctly specify the behaviour of the code implementing the action. Note, that in addition to changing the state of the agent, an action may change the state of other agents' msgboxes.

Example 4.15

Here is an example of a timed action **drive()** of the **truck** agent which may be described via the following components:

Name: **drive**(From, To, Highway)

Schema: (String,String,String)

Pre: **in**(From, **truck**:**location**())

Dur: {T | **in**(X, **math**:**distance**(From, To))
 & **in**(T, **math**:**compute**($\frac{60X}{70}$)) }

Tet: 1st arg : rel: {20}

2nd arg : {**in**(NewPosition, **truck**:**location**(X_{now})) }

3rd arg : {**in**(OldPosition, **truck**:**location**(X_{now} - 20)) }

The **Tet** part says that the **truck** agent updates its location every 20 minutes (assuming a time period is equal to 1 minute) during the expected time it takes it to drive the distance between `From` to `To` at 70km per hour.

Semantics and Verification

One main feature of *IMPACT* is the precise, formal semantics based on the notion of **agent programs**.

These programs are from an abstract point of view, logic programs (**if-then-else** rules).

The semantics of such programs has been investigated extensively in the last three decades.

Each agent has

- 1 a set of **integrity constraints** *IC*—only states that satisfy these constraints are considered to be **valid** or **legal** states
- 2 a notion of **concurrency** specifying how to combine a set of actions into a single action
- 3 a set of **action constraints** that define the circumstances under which certain actions may be concurrently executed
- 4 an **agent program** that determines what actions the agent can take / cannot take / must take. Agent programs are defined in terms of status atoms defined below.

Definition 4.16 (Status Atom/Status Set)

If $\alpha(\vec{t})$ is an action, and $Op \in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{Do}, \mathbf{O}\}$, then $Op\alpha(\vec{t})$ is called a **status atom**.

If A is a status atom, then $A, \neg A$ are called **status literals**.

A **status set** is a finite set of ground status atoms.

Intuitively:

P α means α is permitted

F α means α is forbidden

O α means α is obligatory

Do α means α is to be done

W α means that the obligation to perform α is waived

- Note that these operators are **not independent** from each other.
- For example, an action α cannot have the status **F** and **O** at the same time. And **O** α should always imply **Do** α .
- These interrelations are taken into account by the semantics.

Definition 4.17 (Agent Program)

An **agent program** \mathcal{P} is a **finite set of rules** of the form $A \leftarrow \chi \& L_1 \& \dots \& L_n$, where

- χ is a **code call condition** (χ)
- L_i are **status literals**
- A is a **status atom**.

Definition 4.18 (Temporal Agent Rule/Program \mathcal{TP})

- A **temporal agent rule** is an expression of the form $Op\alpha : [tai_1, tai_2] \leftarrow \rho_1 : ta_1 \& \dots \& \rho_n : ta_n$, where $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, and $\rho_1 : ta_1, \dots, \rho_n : ta_n$ are tascs
- (A tasc (temporal action status conjunct) is, intuitively, a conjunction of temporal status actions).
- A **temporal agent program (tap)** is a finite set of temporal agent rules.

Intuitive Reading of Temporal Agent Rule

“If for all $1 \leq i \leq n$, there exists a time point t_i such that ρ_i is true at time t_i such that $t_i \in ta_i$ then $Op\alpha$ is true at some point $t \geq t_{now}$ (i.e., now or in the future) such that $tai_1 \leq t \leq tai_2$ ”.

How can taps be used to express the statement in the Rescue Scenario I?

We use two relational databases:

- **shipdata** containing at least the attributes `shiptime`, `orig`, `dest` (and perhaps other ones as well) which specifies data (such as shipping time) associated with past shipments.
- **sched** which has at least the attributes `reqtime`, `place`, `item` specifying which items are required at what time by what places.

► Rescue Scenario I

Example 4.19

```
Do ship(P,A,B):[T - T1 - 10, T - T1] ←  
  (in(T1, db:sql('StimeFdataWorig = A&dest = B')) &  
  in(T, db:sql('SreqtimeFplaceWitem = P'))): [Xnow, Xnow].
```

S: SELECT

F: FROM

W: WHERE

Example 4.20

“If a prediction package expects a stock to rise $K\%$ after T_K units of time and $K \geq 25$ then buy the stock at time $(x_{\text{now}} + T_K - 2)$.”

We assume a prediction package that given a stock uses some stock expertise to predict the change in the value of the stock at future time points. This function **returns a set of pairs of the form (T, C)** . Intuitively, this says that T time units from now, the stock price will change by C percent (positive or negative).

Example 4.21

Finally, here is a tap using several rules and different status atoms.

- ① **Fdrive**(was, bal, hw295): $[t_{\text{now}}, t_{\text{now}} + 2] \leftarrow$
in(hw295, **msgbox**:**gatherWarning(comc)**): $[t_{\text{now}} - 3, t_{\text{now}}]$
- ② **Do fill_fuel**(): $[t_{\text{now}}, t_{\text{now}}] \leftarrow$
in(true, **truck**:**tank_empty()**): $[t_{\text{now}} - 2, t_{\text{now}}]$
- ③ **Oorder_item**(fa_bag): $[t_{\text{now}}, t_{\text{now}} + 4] \leftarrow$
in(1, **truck**:**inventory**(fa_bag)): $[t_{\text{now}} - 3, t_{\text{now}}]$
- ④ **Pdrive**(was, bal, hw95): $[t_{\text{now}}, t_{\text{now}}] \leftarrow$
in(false, **truck**:**tank_empty()**): $[t_{\text{now}}, t_{\text{now}}]$ &
Fdrive(was, bal, hw295): $[t_{\text{now}} + 1, t_{\text{now}} + 2]$

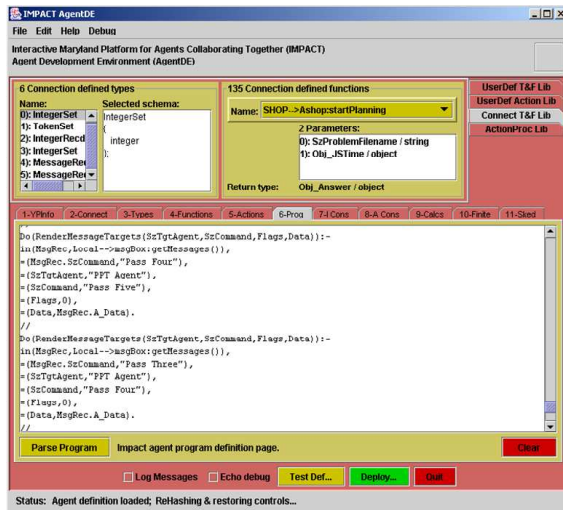


Figure 17: *AgentDE* Program

- Figure ?? shows two rules (with **Do** 's in the head) of the **monitoring** agent in A-SHOP.
- This approach is to base the semantics of agent programs on **consistent** and **closed** status sets.
- **Consistent** means that there are no inconsistencies (such as $\mathbf{F}\alpha$ and $\mathbf{P}\alpha$ in the same set) and **closed** means that when **Do** α is in the set, then so is $\mathbf{P}\alpha$.
- We have to take into account not only the rules of the program but also the **integrity constraints** *IC*. \rightsquigarrow notion of a feasible status set.
- The operator $\mathbf{App}_{\mathcal{P}, O_S}(S)$ is similar to the immediate consequence operator in logic programming: it computes all the **consequences obtainable from applying all agent rules once**.

Definition 4.22 (Feasible Status Set)

Let \mathcal{P} be an agent program, and let O_S be an agent state. Then, a status set S is a **feasible status set** for \mathcal{P} on O_S , if the following conditions hold:

(S1) (closure under the program rules)

$$\mathbf{App}_{\mathcal{P}, O_S}(S) \subseteq S;$$

(S2) (deontic/action consistency)

S is deontically and action consistent;

(S3) (deontic/action closure)

S is action closed and deontically closed;

(S4) (staten consistency)

$O'_S \models IC$, where $O'_S = \text{apply}(\mathbf{Do}(S), O_S)$ is the state which results after taking all actions in $\mathbf{Do}(S)$ on the state O_S .

- The last condition ensures that the successor state (when all doable actions are executed) still satisfies the integrity constraints *IC*.
- The semantics of agent programs is then defined by rational status sets.

Definition 4.23 (Groundedness; Rational Status Set)

A status set S is **grounded**, if there exists no status set $S' \subsetneq S$ such that S' satisfies conditions (S1)–(S3) of a feasible status set.

A status set S is a **rational status set** if S is a feasible status set and S is grounded.

- Given an agent program, our semantics **computes all rational status sets of this program**.
- In the case of positive agent programs it can be shown that there always **exists exactly one rational status set**. Rational status sets are natural generalisations of stable models (or answer sets) in logic programming.

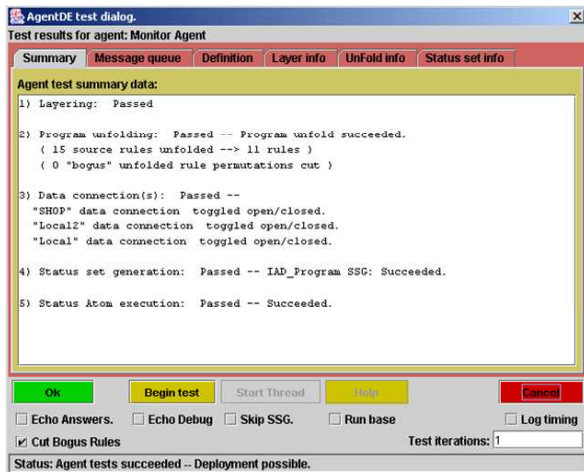


Figure 18: *AgentDE* Summary Table

Other features of the language

Complexity: special emphasis is put on identifying classes of programs that can be efficiently implemented. The class of **regular** agents (based on a special class of agent programs) ensures that its complexity modulo the underlying legacy code is only polynomial.

Legacy code: existing legacy code can be turned into an *IMPACT* agent (**agentisation**). This is illustrated with A-SHOP, which is an agentised version of SHOP, a well-known planning system.

Complexity Issues

Remember the condition of safeness to ensure evaluability of a code call.
Remember also that an evaluable **CC** does not need to terminate.

Example 4.24

Consider the code call

```
in(X, math:geq(25)) &  
in(Y, math:square(X)) & Y ≤ 2000,
```

which constitutes all numbers that are less than 2000 and that are squares of an integer greater than or equal to 25.

- there are only finitely many ground substitutions that cause this code call condition to be true.
- call condition is safe.
- its evaluation may never terminate.

⇒ **In general**, we must impose some restrictions on code call conditions to ensure that they are **finitely evaluable**.

⇒ This is precisely what the condition of **strong safeness** does for the code-call conditions.

Intuitively: By requiring that the code call condition is safe, we are ensuring that it is executable and by requiring that it is strongly safe, we are ensuring that it will only return finitely many answers.

Deciding whether an arbitrary code call execution terminates is **undecidable** ⇒
Finiteness Table

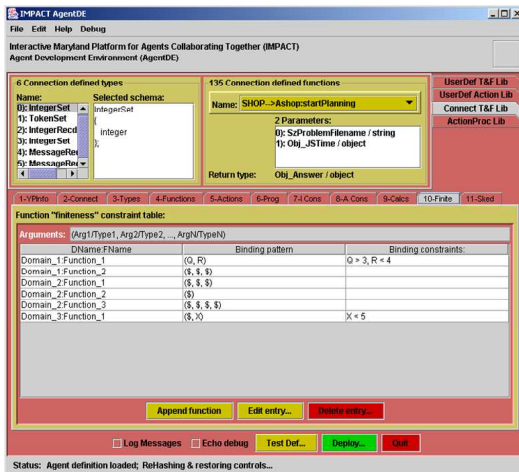


Figure 19: *AgentDE* Finiteness Table

- Information stored in **finiteness table**
- Used in the **purely syntactic** notion of strong safeness
- It is a **compile-time check**, an extension of the well-known (syntactic) safety condition in databases.

Agentisation

Rescue Scenario II illustrate how to turn a planner into an planning agent within a multi-agent environment.

SHOP:

- *HTN* planner (based on the concepts of **tasks**, **operators** and **methods**).
- Methods are used to decompose a nonprimitive task and form the heart of *HTN* planning.

Comparison between *IMPACT*'s actions and SHOP's methods:

- *IMPACT* actions \longleftrightarrow fully instantiated methods (*HTN*)

► Rescue Scenario II

- SHOP's methods and operators are based on STRIPS
- First step is to modify the atoms in SHOP's preconditions and effects, so that SHOP's preconditions will be evaluated by *IMPACT*'s code call mechanism and the effects will change the state of the *IMPACT* agents.
- This is a fundamental change in the representation of SHOP.
- Replacing SHOP's methods and operators with **agentised** methods and operators.

Definition 4.25 (Rescue II, **Agentised Operator**)

An **agentised operator** is an expression of the form **(AgentOp** $h\chi_{add}\chi_{del}$), where h (the **head**) is a primitive task and χ_{add} and χ_{del} are lists of code calls (called the **add-** and **delete-lists**).

The set of variables in the tasks in χ_{add} and χ_{del} is a subset of the set of variables in h .

Lemma 4.26 (Rescue II, **Evaluating Agentised Operators**)

*Let **(AgentOp** $h\chi_{add}\chi_{del}$) be an agentised operator.*

*If the add and delete-lists χ_{add} and χ_{del} are strongly safe wrt. the variables in h , the problem of applying the agentised operator to **O** can be algorithmically solved.*

- In SHOP, **preconditions were logical atoms**, and SHOP would infer these preconditions from its current state of the world using Horn-clause inference.
- In contrast, the preconditions in an agentised method are **IMPACT's code call conditions** rather than logical atoms.
- Also A-SHOP (the agentised version of SHOP) does not use Horn-clause inference to establish these preconditions but instead simply invokes those code calls, which are calls to other agents.
- This opens the way to **use arbitrary reasoning mechanisms** and **data distributed over the net**.

Theorem 4.27 (Rescue Scenario II, **Sound- and Completeness**)

Let O be a state and \mathcal{D} be a collection of agentised methods and operators. If all the preconditions in the agentised methods and add- and delete-lists in the agentised operators are **strongly safe** wrt. the respective variables in the heads, then A-SHOP is **sound** and **complete**.

Head:

AirTransport(LocFrom, LocTo, Cargo, CargoWeight)

Preconditions:

in(CargoPL, **supplier**: *cargoPlane*(LocFrom))&
in(Dist, **statistics**: *distance*(LocFrom, locTo))&
in(DCargoPL, **statistics**: *authorRange*(CargoPL))&
Dist \leq DCargoPL&
in(CCargoPL, **statistics**: *authorCapacity*(CargoPL))&
CargoWeight \leq CCargoPL&

Subtasks:

load(Cargo, LocFrom)
fly(Cargo, LocFrom, LocTo)
unload(Cargo, LocTo)

Figure 20: Agentised method for a logistics problem.

Features of the platform

Agent Development Environment

Agent developers can easily build and test agents within the *IMPACT* Agent Development Environment (*AgentDE*). As described earlier, the core parts of an *IMPACT* agent are:

- 1 a set of data type *API* function calls manipulated by the agent;
- 2 a set of actions that the agent may take;
- 3 a set of integrity constraints *IC* on the agent state and action constraints *AC*;
- 4 an agent program *P* specifying the behaviour of the agent;
- 5 a notion of concurrency *conc*.

AgentDE :

- Network accessible, easy-to-use graphical user interface (for parameters, compiling, testing).
- Contains libraries of data types, *API* functions, actions, notions of concurrency.
- Each data type must be explicitly defined via the *AgentDE*.
- The agent manipulates its data types via *API* function calls (defined within the *AgentDE*).
- The developer needs to specify a set of actions that the agent can execute via *AgentDE*.

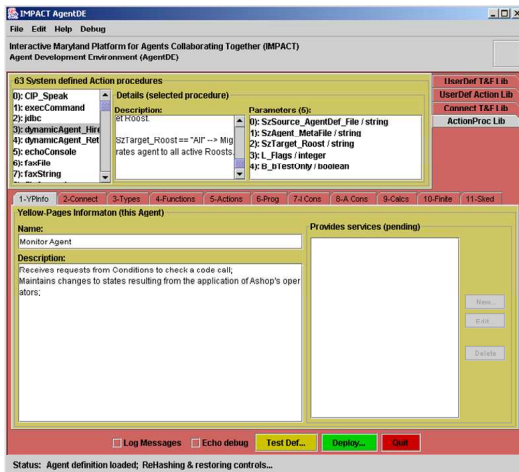


Figure 21: Actions for **monitor** (Reuse actions)

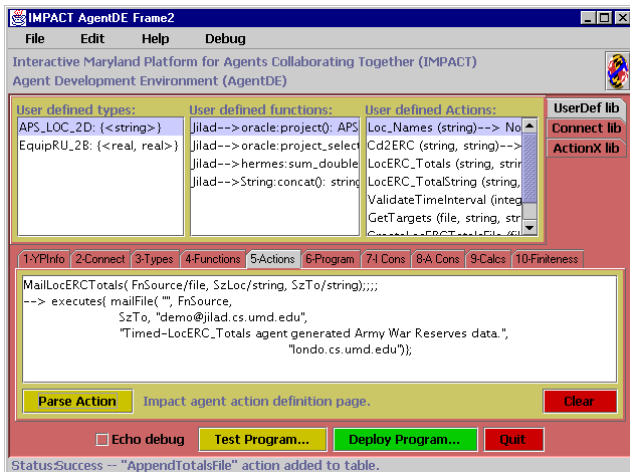


Figure 22: Actions in *AgentDE*

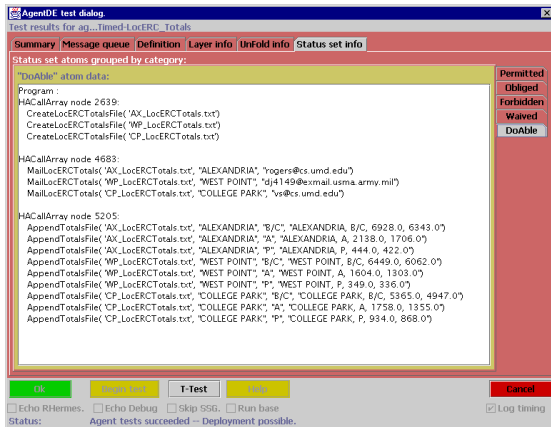


Figure 23: *AgentDE* Status Set Screen

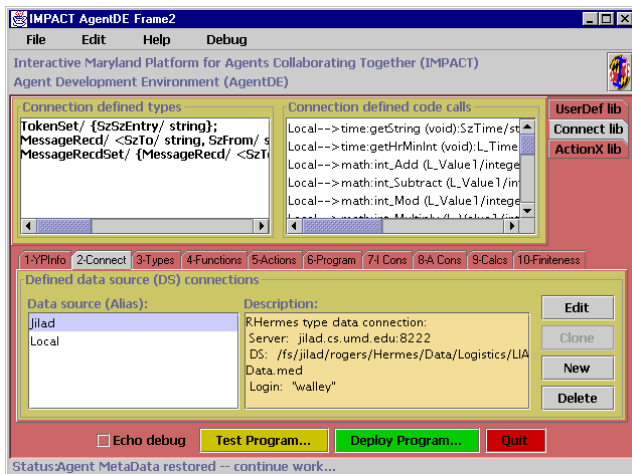


Figure 24: *AgentDE* Connect Library Screen

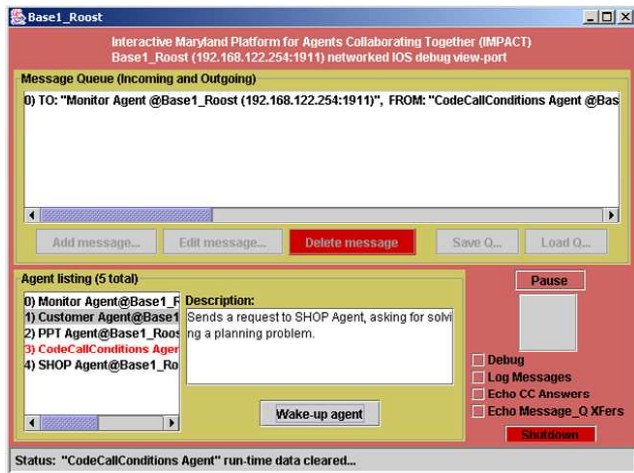


Figure 25: Agent Roost

IMPACT Connection

The *IMPACT* connection library allows *IMPACT* agents to access third party platforms.

IMPACT Server

The *IMPACT* Server provides various services that are required by a group of agents as a whole. It supports the following services:

Registration Services The developer can register the services provided by the agent and also specify who can use those services.

Yellow Pages Services *IMPACT* agents can find the desired services by other agents via the Yellow Pages Server.

Type Services Agent developers can specify the datatypes they use as well as the relationship between the newly created datatypes and other existing types within the *IMPACT* Type Server.

IMPACT Server (continued)

The *IMPACT* Server provides various services that are required by a group of agents as a whole. It supports the following services:

Thesaurus Server This server receives requests when new agent services are being registered and when the *IMPACT* Yellow Pages Server is looking for agents providing a service.

Ontology Services The *IMPACT* server is able to provide ontology services. An agent can reformulate its query in terms the other agent can understand.

Agent Roost

An **agent roost** is a location where a set of deployed agents resides (Figure ?? shows the five agents in A-SHOP: the screen depicts the moment when the **codecallconditions** agent is active and sends a message to the **monitoring** agent). An agent roost serves as a duty officer since it manages all messages for this set of agents.

Agent Log

The **agent log** allows an agent developer to maintain a record of agent communication and agent actions. The log supports log queries by content or time, and action browse, playback of video, text and image message objects.

Available tools and documentation

A tutorial about *IMPACT* can be found at

<http://www.cs.umd.edu/projects/impact>. In particular, there is an *IMPACT* software library user documentation, which is available at <http://www.cs.umd.edu/projects/impact/Docs>, and includes:

- 1 Implementation overview.
- 2 Introduction of agent instantiation life cycle.
- 3 Agent definition syntax.
- 4 Sample agent development.
- 5 Selected user and developer code *API* JavaDocs.

Standards compliance, interoperability and portability

The implementation code consists of three main components:

- The **IMPACT AgentDE** (containing a series of compilers, written in Java, which render an agent instantiation from a given agent definition (text))
- The **IMPACT Yellow-pages server**, written in Java and C, provides agent directory lookup services necessary for agent construction and run-time communication;
- The **IMPACT Roost**, written in Java, provides a run-time environment for *IMPACT* agents to work, sleep, or travel the network.

The *IMPACT* project has built applications in the following areas:

1. US Army Logistics Integration Agency's "Virtual Operations Centre" involves the integration of a wide variety of distributed, heterogeneous databases, together with diverse alert, analysis and visualisation requirements.
2. US Army Research Laboratory's "Combat Information Processor" project where *IMPACT* is used to provide yellow pages matchmaking services, and is also providing alert mechanisms for multiple users with diverse battlefield monitoring requirements.
3. Aerospace applications where *IMPACT* technology has led to the development of a multi-agent solution to the "Controlled Flight into Terrain" problem which is the single largest cause of human fatalities in aircraft crashes (Washington Post, Feb. 7, 1998).

4. US Army STRICOM's JANUS project where *IMPACT* technology is used to analyse massive amounts of simulation data.
5. Coordinated route and flight planning applications over free terrain.

New applications in the banking and finance sector are under consideration. In addition, *IMPACT* has been used for student projects in academia, including University of Maryland, Technical University of Vienna, The University of Manchester, and Clausthal Institute of Technology.

Agent-Oriented Programming Language

Functionality

- Q: Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A: Yes, our language supports **reasoning with beliefs, time, probabilities** and various other concepts (not all of them are yet implemented). Deliberation is realised through **computing (feasible) status sets** and is related to computing stable models.

Communication

- Q: Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A: Yes, speech-act primitives are available, although only **very basic message passing capabilities** are realised in the base language.

Underlying Computational Model

Q: Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?

A: Yes, *IMPACT* supports **weak mobility**.

Simplicity

Q: How easy it is to use and understand the language?

A: We have run several classroom labs with students. They did not have to go through the technical semantics, but were given several examples and learned by analogy. They were able to implement a non-trivial application involving 6 agents (each of them developed independently) and successfully putting them together (**Gofish post office**).

Preciseness

- Q: Does the language have clear and precise semantics? How has it been formalised?
- A: Yes, the semantics is clearly defined and uses technical machinery developed in the last three decades in **logic programming**.

Expressiveness

Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?

A: The language is suitable for arbitrary agent applications. It is not specific for a restricted class of applications.

Extensiveness

- Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?
- A: Yes, the language allows not only the definition of macros of basic constructs, but also the introduction of completely new features. This is due to **annotations of programs**, an area which has been well investigated in the last two decades.

Verification

- Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?
- A: Yes, as our semantics is based on rigorous formal methods and first attempts to verify *IMPACT* agents are on their way.

Software Engineering Principles

Q: Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?

A: no answer.

Language Integration

- Q: Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?
- A: The **Code Call Condition** mechanism provides a way to integrate any software program written in any programming languages.

Language Integration (continued)

- Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?
- A: This can be achieved by the *IMPACT* implementation and the **Code Call Condition** mechanism.

Platform

Deployment and Portability

Q: Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?

A: The *IMPACT* project homepage

(<http://www.cs.umd.edu/projects/impact/>) provides extensive documentations to help users develop and deploy systems.

Deployment and Portability (continued)

Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?

A: No, it runs on any platform where Java is available.

Standards Compliance

- Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?
- A: While *IMPACT* is not FIPA compliant, it should not be too difficult to achieve this. We concentrated on extending our framework and not on compliance to certain standards.

Platform Extensibility

Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?

A: *IMPACT* can be easily extended by new functionalities. Although it is not open source yet, any collaboration is welcome.

Available Tools

Q: What tools are provided by the platform for the management, monitoring, logging and debugging of applications?

A: It is provided by *IMPACT AgentDE*, *IMPACT* Server, Agent Roost, and Agent Log.

Available Tools (continued)

Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?

A: The software user documentation is available at
<http://www.cs.umd.edu/projects/impact/Docs>.

Available Tools (continued)

Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?

A: *IMPACT* provides a network accessible, easy-to-use IDE.

Tool Integration

- Q: In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?
- A: We have shown that the *IMPACT* project has built a lot of applications integrating many tools.

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** Applications require loading Java runtime library on the target platform, but no specific operating system is required.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: We have not undertaken such statistics yet. In our experiments there were around 10-15 agents with quite a number of messages sent among them.

Performance Issues (continued)

- Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?
- A: The current version is stable and available for academic purposes only. It is licensed by the University of Maryland.

Multi-Agent Systems Features

- Q: Does the platform support open multi-agent systems and heterogeneous agents?
- A: Yes, *IMPACT* supports open multi-agent systems and heterogeneous agents.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: *IMPACT* provides both centralised and distributed control. The hierarchical structuring of agents has not yet been supported within *IMPACT*.

Multi-Agent Systems Features (continued)

Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?

A: Yes, they are provided in *IMPACT AgentDE* during the development and deployment of agents.

Applications Supported by the Language and Platform

Typical Examples

- Q:** What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?
- A:** Besides smaller classroom examples (6 agents developed by 4-7 students) several real applications such as Aerospace applications, US Army Logistics Integration Agency's "Virtual Operations Centre", and US Army STRICOM's JANUS project where *IMPACT* technology is used to analyse massive amounts of simulation data.

Targeted Domains

- Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?
- A: There is no specific targeted domain for *IMPACT*. It is a general system dealing with heterogenous, distributed information sources and available legacy code.

References I



Jürgen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang.

Monitoring Agents using Declarative Planning.

Fundamenta Informaticae, 57(2–4):345–370, 2003.



Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian.

Temporal agent reasoning.

Artificial Intelligence, 127(1):87–135, 2001.



Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian.

Heterogenous temporal probabilistic agents.

ACM Transactions of Computational Intelligence, 7(1):151–198, 2006.

References II



Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang.
Planning in a multi-agent environment: Theory and practice.
In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 944–945. New York: ACM Press, July 2002.



Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang.
IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment.
Annals of Mathematics and AI, 37(4):381–407, 2003.



Jürgen Dix, Mirco Nanni, and V. S. Subrahmanian.
Probabilistic agent reasoning.
ACM Transactions of Computational Logic, 1(2):201–245, 2000.



Jürgen Dix, Fatma Öczan, and V.S. Subrahmanian.
Improving performance of heavily loaded agents.
Annals of Math and AI, 41(2-4):339–395, 2004.

References III



Jürgen Dix, V.S. Subrahmanian, and George Pick.

Meta Agent Programs.

Journal of Logic Programming, 46(1-2):1–60, 2000.



T. Eiter and V.S. Subrahmanian.

Heterogeneous Active Agents, II: Algorithms and Complexity.

Artificial Intelligence, 108(1-2):257–307, 1999.



T. Eiter, V.S. Subrahmanian, and T.J. Rogers.

Heterogeneous Active Agents, III: Polynomially Implementable Agents.

Artificial Intelligence, 117(1):107–167, 2000.



Thomas Eiter, V.S. Subrahmanian, and Georg Pick.

Heterogeneous Active Agents, I: Semantics.

Artificial Intelligence, 108(1-2):179–255, 1999.

References IV



H. Muñoz-Avila, D.W. Aha, D.S. Nau, R. Weber, L. Breslow, and F. Yaman.
SiN: Integrating case-based reasoning with task decomposition.
In *IJCAI-2001*, August 2001.



V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus,
Fatma Özcan, and Robert Ross.
Heterogenous Active Agents.
MIT-Press, 2000.

Abstract

JADE (Java Agent Development Framework) is a software environment to build agent systems for the management of networked information resources in compliance with the FIPA specifications for interoperable multi-agent systems. **JADE** provides a middleware for the development and execution of agent-based applications which can seamless work and interoperate both in wired and wireless environment. Moreover, **JADE** supports the development of multi-agent systems through the predefined programmable and extensible agent model and a set of management and testing tools. Currently, **JADE** is one of the most used and promising agent development framework; in fact, it has a large user group, involving more than two thousands active members, it has been used to realize real systems in different application sectors, and its future development is guided by a governing board involving some important industrial companies.

Keywords

Agent development framework, FIPA compliant agent platform, Middleware for heterogeneous networks, Java

Authors

Fabio Bellifemine, Federico Bergenti, Giovanni Caire and Agostino Poggi

- **Agents** are considered one of the most promising **information technologies** to **realize distributed interoperable systems**.
- Agent-based technologies could not keep their promises and become wide-spread, until there were no suitable standards to support agent interoperability and adequate environments for the development of agent systems.
- Different groups of researchers started working towards the definition of standards for agent technologies and the realization of development environments for multi-agent systems.

- **FIPA specifications** and the **JADE software framework** may be considered two of most interesting results in these two fields.
- FIPA specifications define the reference model of an agent platform and a set of services that should be provided to realize truly interoperable multi-agent systems.
- **JADE** (Java Agent Development framework) is a **software environment** to build **agent systems** for the management of networked information resources in compliance with the **FIPA specifications**.

- The **JADE framework** is based on a **middleware** that facilitates the development of distributed multi-agent applications based on a peer-to-peer communication architecture.
- The **intelligence, the initiative, the information, the resources and the control** can be fully distributed on mobile terminals as well as on hosts in the fixed network.
- The **environment** can evolve dynamically with agents that appear and disappear in the system according to the needs and the requirements of the context.
- **Communication** between agents, regardless of whether they are running in the wireless or wireline network, is completely symmetric with each agent is able to play both the initiator and the responder role.

Main principles

JADE is based of the following **main principles**:

1. Interoperability

- JADE is compliant with FIPA specifications.
- As a consequence a JADE agent can interoperate with other peers not running on the JADE run-time.

2. Uniformity and portability

- JADE provides applications with a homogeneous set of APIs that are independent from the underlying network and Java version.
- The JADE run-time provides the same APIs both for the J2EE, J2SE and J2ME environment.
- In theory, developers could decide the Java run-time environment at deploy-time.

Main principles

JADE is based of the following **main principles**:

3. Ease of use

- The complexity of the middleware is hidden behind a simple and intuitive set of APIs.

4. Pay-as-you-go philosophy

- Programmers do not need to use all the features provided by the middleware.
- Features that are not used do not require programmers to know anything about them, neither add any computational overhead.

- **JADE** includes both the **libraries of Java classes** required to develop application agents and the **run-time environment** that provides the basic services.
- **Each instance** of the **JADE** run-time is called **container** (since it “contains” agents).
- The **set of all containers** is called **platform** and provides a homogeneous layer that hides completely from agents the complexity and the diversity of the underlying tires.

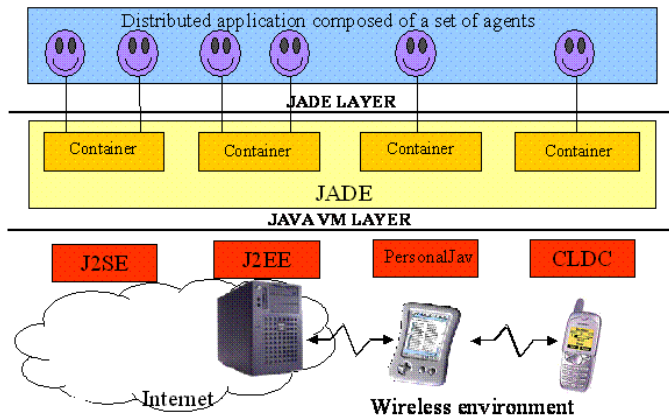


Figure 26: The **JADE** architecture.

Compatibility

- **JADE** framework is compatible with Java J2ME CLDC/MIDP1.0 environment and it has already been tested on the fields over the GPRS network with different mobile terminals.
- The **JADE** run-time memory footprint in a MIDP1.0 environment is around 100 KB, but can be further reduced until 23 KB using the ROMizing technique.
- The limited memory footprint allows installing **JADE** on mostly all cell phones provided that they are Java-enabled.
- **JADE** has already been integrated into complex architectures such as .NET or J2EE.

Properties

- **JADE** provides the basic services necessary to distributed peer-to-peer applications in the fixed and mobile environment.
- **JADE** allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm.
- Each agent is identified by a unique name and provides a set of services.
- It can register and modify its services and/or search for agents providing new services, control its life cycle and, most of all, communicate with all other peers.

Communication

- **Agents communicate** by exchanging **asynchronous messages**.
- In order to communicate, an agent just sends a message to a destination.
- **Agents** are **identified by** a **name**
- There is **no temporal dependency** between communicating agents.
- The sender and the receiver could not be available at the same time.
- The receiver may not even exist or could not be directly known by the sender that can specify, e.g., “all agents interested in football” as a destination.

Security

- Despite this type of communication, **security** is preserved, since **JADE** provides proper mechanisms to **authenticate and verify “rights”** assigned to **agents**.
- **An application** can **verify** the **identity of the sender of a message** and **prevent actions** that it is not allowed to perform.
- All messages exchanged between agents are carried out within an **envelope** including only the information required by the transport layer.
- This allows, among others, to **encrypt the content** of a message separately.

Skeleton

- This development framework provides a set of skeletons of typical interaction patterns associated with specific tasks such as negotiations, auctions and task delegation.
- By using these skeletons programmers can get rid of the burden of dealing with synchronization issues, timeouts, error conditions.

Conversion

- To facilitate the creation and handling of messages content, **JADE** provides a rich support for automatically converting back and forth between a string formats including **XML** and **RDF** and **Java objects**.
- This support is integrated with existing ontology creation tools allowing programmers to graphically create their ontology and then work with Java objects to handle message contents conformant to it.

Parallel tasks

- To increase scalability or to meet the constraints of environments with limited resources, **JADE** provides the opportunity of executing multiple parallel tasks within the same Java thread.
- Several elementary tasks may then be combined to form more complex tasks structured as concurrent Finite States Machines.

Code and execution-state mobility

- In the J2SE and Personal Java environments, **JADE** supports **code and execution-state mobility**.
- That is an agent can stop running on a host, migrate on a different remote host and restart its execution there from the very point it was interrupted.
- This functionality allows for example to distribute computational load at runtime by moving agents on less loaded machines without any impact on the application.

Naming service, yellow pages service

- The platform also includes a naming service (ensuring each agent has a unique name) and a yellow pages service that can be distributed across multiple hosts.
- Federation graphs can be created to support the definition of agent services domains.

LEAP

- A module called **LEAP** allows optimising all communication mechanisms when dealing with devices with limited resources and connected through wireless networks.
- A **JADE** container is “**split**”, as depicted in figure ??, into a:
 - Front-end, actually running on the mobile terminal;
 - A back-end, running in the fixed network.
- A proper architectural element, **called mediator**, must be already active.
- It is in charge of instantiating and holding the back-ends.
- To face work-load problems it is possible to deploy several mediators..
- Each front-end is linked to its corresponding back-end.

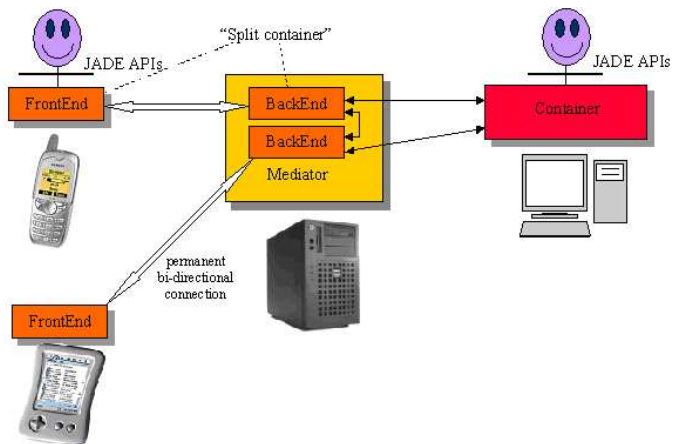


Figure 27: **JADE** architecture in the wireless environment.

Advantages

The described approach has a number of advantages:

- Part of the functionality of a container are delegated to the back-end, thus making the front-end extremely lightweight in terms of required memory and processing power.
- The back-end masks the actual IP address assigned to the wireless device to other containers.
- The front-end is able to detect a loss of connection with the back-end and to re-establish it as soon as possible.
- Both the front-end and the back-end use a store-and-forward mechanism to buffer messages, that cannot be transmitted due to a temporary disconnection, and then deliver as soon as the connection is re-established.
- Several information that containers exchange are handled only by the back-end.

Available tools and documentation

- **JADE** offers a set of documents (manuals and tutorials) and code examples to help users to install and use it; they are all available from the official **JADE** Web site.
- **JADE** provides a rich suite of graphical tools supporting both the debugging, management and monitoring phases of the application life cycle. (figure ?? shows their graphical interfaces).

It is possible to:

- Emulate remote conversations, “sniff” messages exchanged by agents, monitor tasks executed by a specific agent and view its life cycle;
- Control agents running in the system, start, suspend and terminate agents even on remote hosts, inspect and modify the services published in the yellow pages and generate suitable logs.

All these tools are implemented as agents themselves. They require no special support to perform their tasks and they simply rely on **JADE** AMS (Agent Management System).

Remote Management Agent (RMA)

- The general management console for a **JADE** agent platform is called **RMA (Remote Management Agent)**.
- The RMA acquires information about the platform and executes the GUI commands to modify the status of the platform through the AMS.
- It asks the AMS to be notified about changes of state of platform agents.
- It transmits to the AMS the requests for creation, deletion, suspension and restart received by the user.
- The Directory Facilitator agent also has a GUI of its own, with which the DF can be administered, adding or removing agents and configuring their advertised services.

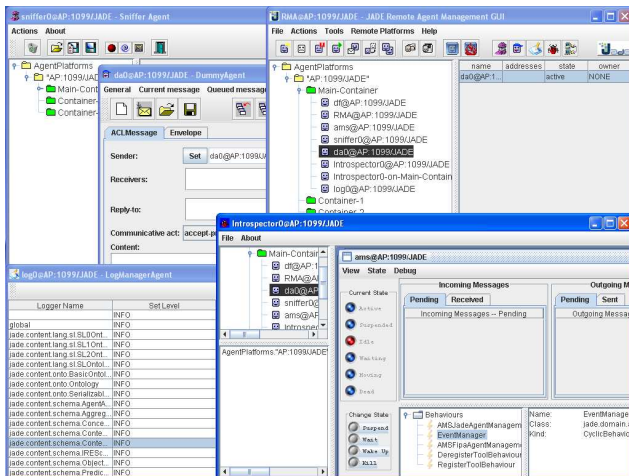


Figure 28: Graphical interfaces of **JADE** tools.

Graphical tools

The graphical tools with which **JADE** users can debug their agents are:

- The **Dummy Agent**;
- The **Sniffer Agent**;
- The **Introspector Agent**;
- The **Log Manager Agent**.

Dummy Agent

- The Dummy Agent is a simple, yet very useful, tool for inspecting message exchanges among agents.
- The Dummy Agent facilitates validation of an agent message exchange pattern before its integration into a multi-agent system and facilitates interactive testing agents.
- The graphic interface provides support to edit, compose and send ACL messages to agents, to receive and view messages from agents, and, eventually, to save/load messages to/from disk.

Sniffer Agent

- The Sniffer Agent makes it possible to track messages exchanged in a **JADE** agent platform.
- When the user decides to sniff a single agent or a group of agents, every message directed to or coming from that agent or group of agents is tracked and displayed in the sniffer window, using a notation similar to UML Sequence Diagrams.
- Every ACL message can be examined by the user, that can also save and load every message track for later analysis.

Introspector Agent

The Introspector Agent is a very powerful tool that allows to debug and introspect a running agent through the following functionalities:

- Monitor and control the agent life-cycle;
- Inspect all its exchanged messages, both the queue of sent and received messages;
- Monitor the queue of behaviours, including the possibility of executing a behaviour step-by-step, in a similar way to a code debugger.

Log Manager Agent

- The Log Manager Agent tracks and stores events happening during the life of multi-agent systems.

Add-ons

- Different **"add-ons"** to the **JADE** framework are provided by both the **JADE** team and other **JADE** users.
- Tools and software libraries have been developed and made available for the management of:
 - The persistence of agents;
 - The security of systems;
 - The exchange of messages and the management of ontologies.
- Add-ons are also available for the integration of **JADE** with other technologies, such as Servlets, JSP, Applets, the JESS rule engine, and the Protege ontology tool.

Standards compliance, interoperability and portability

- **JADE** is available on all the Java versions from J2EE to J2ME.
- **JADE** facilitates the porting of applications among different Java versions.
- **JADE** participated to the bake-off organized by FIPA.

JADE allows:

- The bi-directional communication between agents acting on different wired and wireless networks;
- To build agent systems in compliance with FIPA specifications;
- The interoperability with agents acting on different agent platforms with the constraint that they must be FIPA-compliant.

Other features of the platform

- The whole **JADE** source code is distributed under the Lesser GNU Public License **LGPL**.
- **TILAB**, as project initiator, holds the exclusive right to re-release **JADE** under different or additional license terms.
- The different releases of the **JADE** software are stable and used in different research and application projects in different part of world.
- Different agent development platforms derive from **JADE** (**JADEX**, **BlueJADE**).

- A **JADE** system is based on a set of agent platforms each of them composed of a set of agent containers deployed on an heterogeneous network.
- Message exchanges between agents is managed efficiently using different techniques for intra- and inter-container communication.
- a **JADE** system may contain thousands of agents exchanging a huge amount of messages.

- A **JADE** system provides a centralized control.
- Following the FIPA standard, each **JADE** platform is controlled by the AMS.
- **JADE** offers a **fault tolerance mechanism** that allows an agent platform to survive the failure of its AMS.
- Each agent platform may have different AMSs usually on different containers:
 - One is active;
 - The others are in backup ready to replace it when it fails.

Realizing multi-agent systems with the platform

- A **JADE** multi-agent application is composed of the FIPA standard agents, provided by the **JADE** platform, and of a set of application dependent agents realized by the application developer.
- Agents are implemented through a Java class containing a set of inner classes that realize the different behaviours of the agent.
- Agent behaviours can be composed of other behaviours and can be executed either a single time (one-shot behaviours) or different times (cyclic behaviours).

Agent classes are based on a method:

- **setup**, that performs the agent initialization;
- **takedown**, that performs clean-up operations at the end of its execution.

Agent behaviours are based on a method:

- **action**, that defines the operations to be performed when the behaviour is in execution.

Cyclic behaviours may have another method:

- **done**, that returns a Boolean value indicating whether or not this behaviour has completed its iterative execution.

Code of an agent class

```
public class AgentClassName extends Agent {
    ... variables definition ...
    protected void setup() {
        ... initialize the agent ...
    }
    protected void takeDown() {
        ... clean-up operations ...
    }
    private class RBehaviourClassName extends Behaviour {
        ... variables definition ...
        public void action() {
            ... behaviour execution ...
        }
        // optional method for cyclic behaviours
        public boolean done() {
            ... return true if execution is completed
        }
    }
    ... other behaviour inner classes ...
}
```

Example: Simple book trading multi-agent system

- This system is based on some seller and buyer agents.
- Each buyer agent receives the title of the book to buy as a command line argument from its user and periodically requests all known seller agents to provide an offer.
- As soon as an offer is received, the buyer agent accepts it and issues a purchase order.
- If more than one seller agent provides an offer the buyer agent accepts the best one (lowest price).
- Having bought the target book the buyer agent terminates.
- Each seller agent has a minimal GUI (insert new titles, associated price).
- Seller agents continuously wait for requests from buyer agents.

- When asked to provide an offer for a book, they check if the requested book is in their catalogue, and in this case reply with the price; otherwise they refuse.
- When they receive a purchase order they serve it and remove the requested book from their catalogue.
- A buyer agent is implemented by the **BookBuyerAgent** class.
- Given that a buyer agent has the only goal of buying books on the behalf of its user, then it needs to realize a single behaviour implemented by the **RequestPerformer** inner class.
- This behaviour has to send a call for proposal (CFP) message to the known seller agents, get back all the replies and, in case at least a reply proposing a bid for the request is received, send a further message accepting the proposal to the seller agent that made the best proposal and, finally, get back the response.

- A seller agent is implemented by the **BookSellerAgent** class.
- This agent needs to wait for book requests from buyer agents and serves them; these requests can be requests to provide an offer for a book or purchase orders.
- This is done through two different cyclic behaviours:
 - One dedicated to serve offer requests, implemented by the **OfferRequestsServer** inner class;
 - The other dedicated to serve purchase orders, implemented by the **PurchaseOrdersServer** inner class.
- The seller agent needs to execute a one-shot behaviour updating the catalogue of books available for sale whenever its user adds a new book from the GUI.
- This is done through a method, called **updateCatalogue**, that creates the behaviour the agent needs and adds it to the list of running behaviours.

RequestPerformer

```
public void action() { // RequestPerformer
    switch (step) {
        case 0: // Send the cfp to all sellers
            ... CFP message construction ...
            myAgent.send(cfp);
            ... message type reception setting ...
            step = 1;
            break;
        case 1: // Receive all proposals/refusals
            ACLMessage reply = myAgent.receive(mt);
            if (reply != null) { // Reply received
                ... update the best offer ...
            }
        }
        repliesCnt++;
        if (repliesCnt >= sellerAgents.length) {
            // We received all replies
            step = 2;
        }
        else { block(); }
        break;
    }
}
```

RequestPerformer (continued)

```
case 2: // Send the purchase order
... order message construction ...
myAgent.send(order);
... message type reception setting ...
step = 3;
break;
case 3: // Receive the purchase order reply
reply = myAgent.receive(mt);
if (reply \relax != null) { // Purchase order reply received
    if (reply.getPerformative() == ACLMessage.INFORM) {
        ... inform the user about success and exit ...
    }
    else {
        ... print: requested book already sold ...
    }
    step = 4;
}
else { block(); }
break;
}
}
```

OfferRequestsServer

```
public void action() { // OfferRequestsServer
    ... message type reception setting ...
    ACLMessage msg = myAgent.receive(mt);
    if (msg \relax != null) { // CFP Message received. Process it
        String title = msg.getContent();
        ACLMessage reply = msg.createReply();
        Integer price = (Integer) catalogue.get(title);
        if (price \relax != null) {
            // The requested book is available: reply with the price
            reply.setPerformative(ACLMessage.PROPOSE);
            reply.setContent(String.valueOf(price.intValue()));
        }
        else { // The requested book is not available
            reply.setPerformative(ACLMessage.REFUSE);
            reply.setContent("not-available");
        }
        myAgent.send(reply);
    }
    else { block(); }
}
```

PurchaseOrdersServer

```
public void action() { // PurchaseOrdersServer
    ... message type reception setting ...
    ACLMessage msg = myAgent.receive(mt);
    if (msg \relax != null) {
        // ACCEPT_PROPOSAL Message received. Process it
        String title = msg.getContent();
        ACLMessage reply = msg.createReply();
        Integer price = (Integer) catalogue.remove(title);
        if (price \relax != null) {
            reply.setPerformative(ACLMessage.INFORM);
            ... print: information about the sold book ...
        }
    }
    else {
        // The requested book has been already sold
        reply.setPerformative(ACLMessage.FAILURE);
        reply.setContent("not-available"); }
        myAgent.send(reply);
    }
    else { block(); }
}
```

- **JADE** is being used in a plethora of projects and applications, both from the academic and the industrial communities.
- **JADE** applications cover different domains:
 - Collaborative work support;
 - E-learning;
 - E-tourism;
 - Network management;
 - Entertainment;
 - Knowledge management;
 - Manufacturing and supply-chain management;
 - Simulation.

CoMMA

- CoMMA (Corporate Memory Management through Agents) is a FIPA compliant multi-agent system for the management of a corporate memory, implemented by using JADE .
- It is the result of an international project funded by the European Commission.
- The project started in January 2000 and ended in January 2002.
- The CoMMA system was completely implemented and tested in different companies to offer a helping service for enhancing the insertion of new employees and as a support system for technology monitoring.

- The **innovative aspect** of the system is the integration of several **emerging technologies**.
- These technologies are:
 - Agent technology;
 - Knowledge modelling;
 - XML technology;
 - Information retrieval techniques;
 - Machine learning techniques.

CoMMA developers decided to use agents:

- For wrapping information repositories defining the corporate memory;
- For the retrieval of information;
- For enhancing scaling flexibility and extensibility of the corporate memory;
- To adapt the system interface to the users.

- The use of **JADE** increases system modularity and flexibility.
- The separation between the software platform infrastructure managing agent life-cycle, distribution and communication and the software implementing agent tasks decouples modifications in these two parts.
- The behaviour based agent model, that JADE offers, allows to separate the software code realizing the different tasks of the agents.
- The modification of a task or the introduction of new tasks usually do not cause the modification of other parts of agent code.

- The CoMMA system aims at helping users in the management of an organization corporate memory and in particular at facilitating the creation, dissemination, transmission and reuse of knowledge in an organization.
- The services offered by the CoMMA system are the result of three main tasks:
 - Insertion of XML annotations of new or updated documents;
 - Search of existing documents;
 - Autonomous document delivery in a push fashion to provide her/him with information about new interesting documents.

These tasks are performed through the cooperation among different kinds of agents that can be divided in four sub-societies:

- Document and annotation management;
- Ontology (enterprise and user models) management;
- User management;
- Agent interconnection and matchmaking.

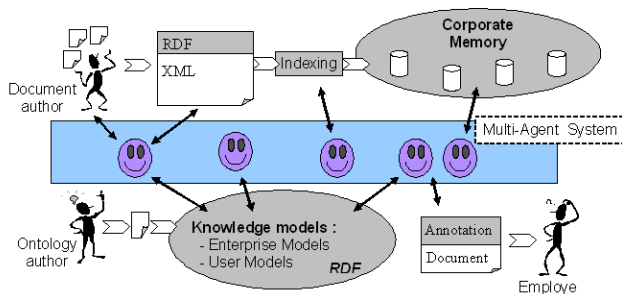


Figure 29: Schematic view of the CoMMA multi-agent system.

Agentcities

- **Agentcities** is a network of FIPA compliant agent platforms that constitute a distributed environment to demonstrate the potential of autonomous agents.
- It started on the second half of 2001 as a research project funded by the European Commission.
- One of the aims of the project is the development of a network architecture to allow the integration of platforms based on different technologies and models.
- It provides **white pages** and **yellow pages services** to allow the dynamic discovery of hosted agents and the services they offer.

- An important outcome is the exploitation of the capability of agent-based applications to adapt to rapidly evolving environments.
- This is particularly appropriate to dynamic societies where agents act as buyers and sellers negotiating their goods and services, and composing simple services offered by different providers into new compound services.
- To allow the integration of different applications and technologies in open environments, high level communication technologies are needed.
- The project largely relies on semantic languages, ontologies and protocols in compliance with the FIPA standards.

- The **Agentcities** network grows around a backbone of 14 agent platforms, mostly hosted in Europe.
- These platforms are deployed as a testbed, hosting the services and the prototype applications developed during the lifetime of the project.
- The backbone is an important resource for other organizations, even external to the project, that can connect their own agent-based services, making the network really open and continuously evolving.

- Currently, the Agentcities network counts 160 registered platforms.
- The platforms are based on more than a dozen of heterogeneous technologies, including **Zeus**, **FIPA-OS**, **Opal**.
- More than 2/3 of them are based on **JADE** and its derived technologies, as **LEAP** and **BlueJADE**.

- The main rationale for using agents is their ability to adapt to rapidly evolving environments and yet being able to achieve their goals.
- In many cases, this can only be accomplished by collaborating with other agents and leveraging on services provided by cooperating agents.

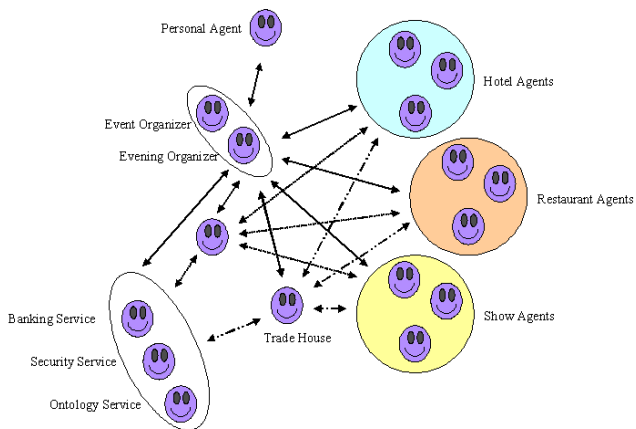


Figure 30: Event organizer scenario.

RAP

- **RAP (Remote Assistant for Programmers)** is a multi-agent system that integrates information and expert searching facilities for communities of students and researchers working on related projects developed in Java.
- **RAP** associates a personal agent with each user, which helps her/him to solve problems proposing information and answers, extracted from some information repositories, and forwarding answers received by “**experts**” recommended on the basis of their expertise on the topic.
- A personal agent also maintains a profile of its user.
- This profile contains information about the competences and experience of its user and is built by using the answers sent to other users and the code written by the user.

The **RAP** system is based on seven different kinds of agents:

- Personal Agents;
- Code Documentation Managers;
- Answer Managers;
- User Profile Managers;
- Email Managers;
- Starter Agents;
- Directory Facilitators.

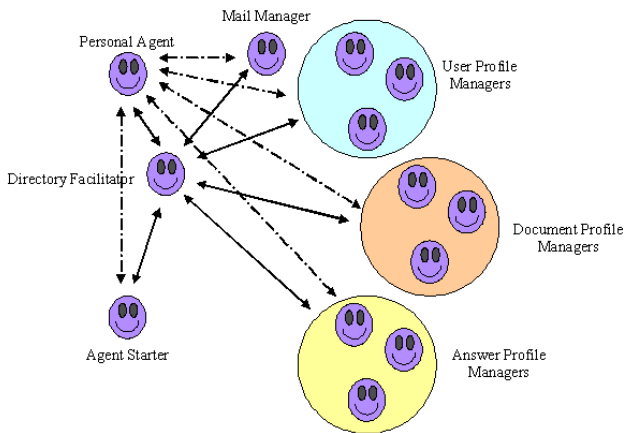


Figure 31: The RAP architecture.

A quite complete description of the behaviour of the system can be given showing the scenario where:

- A user asks information to its personal agent to solve a problem in its code;
- The personal agent finds one (or more) pieces of information that may help her/him.

The description of this scenario can be divided in the following steps:

- 1) Select answer types.
- 2) Submit a query. The query is composed of two parts:
 - “Annotation”;
 - Textual contents.
- 3) Find answers. The reception of new answers from the system users is a more complex activity and its description can be divided in four further steps:
 - 3.1) Find experts.
 - 3.2) Receive expert rating.
 - 3.3) Select experts.
 - 3.4) Receive answers.
- 4) Rate answers.

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** **JADE** provides a very general but primitive agent model offering both reactive and social abilities. This model can serve as a useful basis to implement more sophisticated agent architectures.

Communication

- Q: Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A: **JADE** provides high level communication through FIPA ACL messages. Moreover, it uses different low level communication mechanisms to improve performance.

Underlying Computational Model

Q: Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?

A: **JADE** supports mobile agents through a sort of enhanced weak mobility that allows an agent to move from a node to another node only when its execution reaches a stable state.

Simplicity

Q: How easy it is to use and understand the language?

A: This criteria is not appropriate, because **JADE** does not offer an own language, but software libraries that allow the development of multi-agent systems through the use of Java.

Preciseness

- Q: Does the language have clear and precise semantics? How has it been formalised?
- A: This criteria is not appropriate for the same reason expressed in Simplicity.

Expressiveness

Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?

A: **JADE** has been developed and used to realize systems for different application domains.

Extensiveness

Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?

A: This criteria is not appropriate for the same reason expressed in Simplicity.

Verification

- Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?
- A: The current implementation of **JADE** does not offer any support for the formal verification of programs developed by using the **JADE** software libraries.

Software Engineering Principles

- Q:** Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?
- A:** **JADE** offers the same software engineering and programming language principles offered by the programming language used to implement it (i.e., Java).

Language Integration

Q: Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?

A: **JADE** and the multi-agent systems developed with it are written by using Java.

Language Integration (continued)

- Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?
- A: The agents of a **JADE** multi-agent system can interact with software written in other programming languages by using either special agent wrappers (in the case of non-agentized software) or messages exchange (in the case of other FIPA compliant agents).

Platform

Deployment and Portability

- Q:** Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A:** **JADE** provides a rich set of documents (manuals and tutorials) and code examples to help the user to install and use it. They are all available from the official **JADE** Web site (<http://jade.tilab.com>).

Deployment and Portability (continued)

Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?

A: **JADE** is written in Java. Therefore **JADE** multi-agent systems may run on the operating systems for which a Java virtual machine is available. In particular, the **JADE** run-time can be compiled for different Java profiles allowing the execution of **JADE** multi-agent systems on a wide class of devices ranging from servers to cell phones.

Standards Compliance

Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?

A: **JADE** is FIPA compliant.

Platform Extensibility

- Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?
- A: The whole **JADE** source code is distributed under the LGPL open source licence. Therefore both the extension of the platform and its use in commercial products are allowed.

Available Tools

Q: What tools are provided by the platform for the management, monitoring, logging and debugging of applications?

A: **JADE** users can manage an agent platform through the Remote Management Agent and debug their agents through the Dummy Agent, the Sniffer Agent, the Introspector Agent and the Log Manager Agent. All these agents interact with their users through a graphical user interface.

Available Tools (continued)

Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?

A: **JADE** provides a rich documentation to help the user to install and use it.

Available Tools (continued)

Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?

A: **JADE** users can manage an agent platform through the Remote Management Agent interacting with it through a graphical user interface.

Tool Integration

- Q: In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?
- A: **JADE** permits an easy integration of external software and it was done with success allowing, for example, the integration of **JADE** with: rules engines (JESS and DROOLS), Web technologies (servlets, JSP and applets) and ontology management tools (Protege and Jena).

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** **JADE** multi-agent systems must be written using Java, the only constraint for the operating system is the availability of a Java virtual machine.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: Given its architecture and the different communication mechanisms used, **JADE** multi-agent systems may contain thousands of agents exchanging a huge amount of messages.

Performance Issues (continued)

- Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?
- A: The different releases of the **JADE** software (including the last one: 3.2) are stable and used in different research and application projects in different part of the world. **JADE** is distributed under the LGPL open source licence.

Multi-Agent Systems Features

- Q: Does the platform support open multi-agent systems and heterogeneous agents?
- A: **JADE** allows the realization of open systems through the dynamic federation of agent platforms. Agents of such federations may be heterogeneous with the only constraint of being FIPA compliant.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: Following the FIPA standard, **JADE** multi-agent systems use a centralized control: each agent platform is controlled by the AMS. However, **JADE** offers a fault tolerance mechanism that allows an agent platform to survive the failure of its AMS.

Multi-Agent Systems Features (continued)

- Q:** Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?
- A:** Given that **JADE** agent system are realized by using Java, all its components are reusable. Moreover, the **JADE** framework and its community of users made available different software libraries and “add-ons” that may be useful to realize agent systems in different application sectors.

Applications Supported by the Language and Platform

Typical Examples

Q: What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?

A: **JADE** has been used to realize both real and industrial applications.

Targeted Domains

Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?

A: **JADE** applications cover different domains: collaborative work support, e-learning, e-tourism, network management, entertainment, knowledge management, manufacturing and supply-chain management and simulation.

References I



Fabio BELLIFEMINE.
Special Issue on JADE.
EXP in Search of Innovation, 3(3), 2003.



Fabio BELLIFEMINE, Agostino POGGI, and Giovanni RIMASSA.
Developing multi-agent systems with a FIPA-compliant agent framework.
Software Practice and Experience, 31(2):103–128, 2001.



Michael BERGER, Steffen RUSITSCHKA, Dmitri TOROPOV, Michael WATZKE, and Marc SCHLICHTE.
The Development of the Lightweight Extensible Agent Platform.
EXP in Search of Innovation, 3(3):32–41, 2003.

References II



Brendan Berney and Elaine Ferneley.

CASMIR - A Community of Software Agents Collaborating in Order to Retrieve Multimedia Data.

In Proc. of the third annual conference on Autonomous Agents, pages 428–429, 1999.



Phil Buckle, Tom Moore, Steve Robertshaw, Alan Treadway, Sasu Tarkoma, and Stefan Poslad.

Scalability in Multi-agent Systems: The FIPA-OS Perspective.

In Mark d’Inverno, Michael Luck, Michael Fisher, and Chris Preist, editors, Proc. Foundations and Applications of Multi-Agent Systems, volume 2403 of *Lecture Notes in Computer Science*, pages 110–130. Springer, 2002.



Keith Decker and Katia P. Sycara.

Intelligent adaptive information agents.

Journal on Intelligent Information Systems, 9(3):239–260, 1997.

References III



Fabien Gandon, Agostino Poggi, Giovanni Rimassa, and Paola Turci.
Multi-Agent Corporate Memory Management System.
Applied Artificial Intelligence, 16(9-10):699–720, 2002.



Marco Mari, Alessandro Negri, Agostino Poggi, and Paola Turci.
Agent-Based Remote Assistance for Software Programming Activities.
In *Proc. AI*IA Workshop on Artificial Intelligence & E-Learning*, Perugia, Italy, 2004.



Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis.
ZEUS: A Toolkit for Building Distributed Multiagent Systems.
Applied Artificial Intelligence, 13(1-2):129–185, 1999.

References IV



Martin Purvis, Mariusz Nowostawski, and Stephen Cranefield.

A multi-level approach and infrastructure for agent-oriented software development.

In Proc. of the first international joint conference on Autonomous agents and multiagent systems, pages 88–89, Bologna, Italy, 2002.



Auguste Rabarijoana, Rose Dieng, and Olivier Corby.

Exploitation of XML for Corporate Knowledge Management.

In Dieter Fensel and Rudi Studer, editors, EKAW, volume 1621 of Lecture Notes in Computer Science, pages 373–378. Springer, 1999.

Abstract

This chapter presents **Jadex**, a software framework for the creation of goal-oriented agents following the belief-desire-intention (BDI) model. The **Jadex** project aims to make the development of agent based systems as easy as possible without sacrificing the expressive power of the agent paradigm. The objective is to build up a rational agent layer that sits on top of a middleware agent infrastructure and allows for intelligent agent construction using sound software engineering foundations. Fostering a smooth transition from traditional distributed systems to the development of multi-agent systems, well established object-oriented concepts and technologies such as Java and XML are employed wherever applicable. Moreover, the **Jadex** reasoning engine tries to overcome traditional limitations of BDI systems by introducing explicit goals. This allows goal deliberation mechanisms being realized and additionally facilitates application development by making results from goal-oriented analysis and design easily transferable to the implementation layer. The system is freely available under LGPL license and provides extensive documentation as well as illustrative example applications.

Keywords

BDI agents, FIPA standard, object-oriented software engineering, explicit goals.

Authors

Alexander Pokahr, Lars Braubach and Winfried Lamersdorf

- A numerousness of different agent platforms is available for developing multi-agent applications.
- Most of these platforms are developed with a **specific technological focus** such as the cognitive or infrastructural architecture.
- Not all aspects of agent technology are covered equally well.
- **General applicability of an agent platform** for a great variety of domains demands that at least **three categories of requirements** are considered:
 - Openness;
 - Middleware;
 - Reasoning.

- The existing platforms can be classified into two almost distinct groups:
 - **FIPA-compliant platforms** mainly address openness and middleware issues by realizing the FIPA communication respectively platform standards;
 - **Reasoning-centered platforms** exist, that focus on the behaviour model of a single agent.
- This **gap between middleware and reasoning-centered systems** is one main motivation for the realization of the **Jadex BDI** (Belief-Desire-Intention) reasoning engine.

The **design of the system** is driven by **two main factors**:

- **The development of the reasoning engine** is accompanied by an ongoing effort of enhancing the **BDI architecture** in general. The system addresses shortcomings of earlier BDI agent systems (explicit representation of goals, goal deliberation mechanisms).
- The system respects the current state of the art regarding mainstream **object-oriented software engineering**. **Agent development** is based on established techniques such as **Java and XML**, and is further supported by software engineering aspects, such as reusable modules and development tools.

BDI Models and Systems

- The **BDI model** was initially conceived by Bratman as a **theory of human practical reasoning**.
- Its success is based on its simplicity reducing the explanation framework for complex human behavior to the **motivational stance**.
- In this model, **causes for actions** are only related to **desires** ignoring other facets of cognition such as emotions.
- Another strength of the **BDI model** is the **consistent usage of folk psychological notions** that closely correspond to the way **people communicate about human behavior**.

- The **BDI theory** of Rao and Georgeff defines **belief**, **desires**, and **intentions** as **mental attitudes** represented as **possible world states**.
- The **intentions** of an agent are **subsets** of the **beliefs** and **desires**.
- **Beliefs** are **represented explicitly**.
- **Desires** are **reduced to events** that are handled by **predefined plan templates**.
- **Intentions** are **represented implicitly** by the **runtime stack of plans** to be executed.

Work on **BDI** can be further subdivided into three categories:

- ① General models for practical reasoning, based on BDI concepts;
- ② Computational models based on the “Intelligent Resource-Bounded Machine Architecture”;
- ③ The computational model employed in the **PRS** family of systems.

Concepts within Jadex

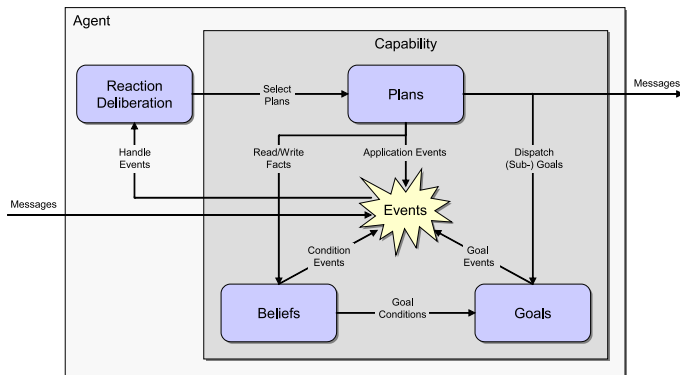


Figure 32: Jadex abstract architecture

Concepts within Jadex

Beliefs

- One objective of the **Jadex** project is the adoption of a software engineering perspective for describing agents.
- In other BDI systems, beliefs are represented in some kind of first-order predicate logic (e.g. **Jason** or using relational models (e.g. JACK and JAM).
- In **Jadex** , an object-oriented representation of beliefs is employed, where:
 - **Arbitrary objects** can be stored as **named facts** (called beliefs);
 - Or **named sets of facts** (called belief sets).
- Operations against the **beliefbase** can be issued in a **descriptive set-oriented query language**.

- The **beliefbase** is not only a passive data store, but takes an active part in the agent's execution, by monitoring belief state conditions.
- Changes of beliefs may therefore directly lead to actions such as events being generated or goals being created or dropped.

Concepts within Jadex

Goals

- **Goals** are a central concept in **Jadex** , following the general idea that goals are **concrete, momentary desires** of an agent.
- For any goal it has, an agent will more or less directly engage into suitable actions, until it considers the goal as being reached, unreachable, or not wanted any more.
- In other PRS-like systems, goals are represented by a special kind of event.
- In these systems the current goals of an agent are only implicitly available as the causes of currently executing plans.

- In **Jadex** , goals are represented as **explicit objects contained in a goalbase**.
- Because goals are represented separately from plans, the system can retain goals that are not currently associated to any plan.
- **Jadex** does not require that all adopted goals are consistent to each other, as long as only consistent subsets of those goals are pursued at any time.
- To distinguish between just adopted and actively pursued goals, a **goal lifecycle** is introduced which consists of the goal states:
 - **Option**;
 - **Active**;
 - **Suspended**.

- When a goal is adopted, it becomes an option that is added to the agent's goalbase, either as top-level goal, or when created from a plan as subgoal of a plan's root goal.
- Application specific goal deliberation settings specify dependencies between goals, and are used for managing the state transitions of all adopted goals (i.e. deciding which goals are active and which are just options).
- Some goals may only be valid in specific contexts determined by the agent's beliefs.
- When the context of a goal is invalid, it will be suspended until the context is valid again.

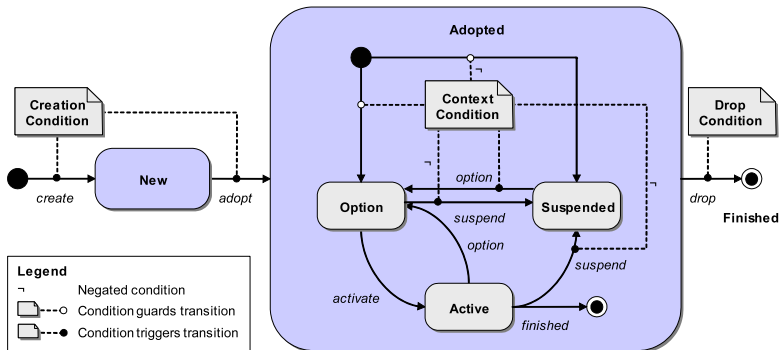


Figure 33: Goal lifecycle

Jadex supports four types of goals:

Perform goal is directly related to the execution of actions.

The goal is considered to be reached, when some actions have been executed, regardless of the outcome of these actions;

Achieve goal is a goal in the traditional sense, which defines a desired world state without specifying how to reach it.

Agents may try several different alternative plans, to achieve a goal of this type;

Query goal The desired state is not a state of the (outside) world, but an internal state of the agent;

Maintain goals An agent keeps track of a desired state, and will continuously execute appropriate plans to re-establish this maintained state whenever needed.

Concepts within Jadex

Plans

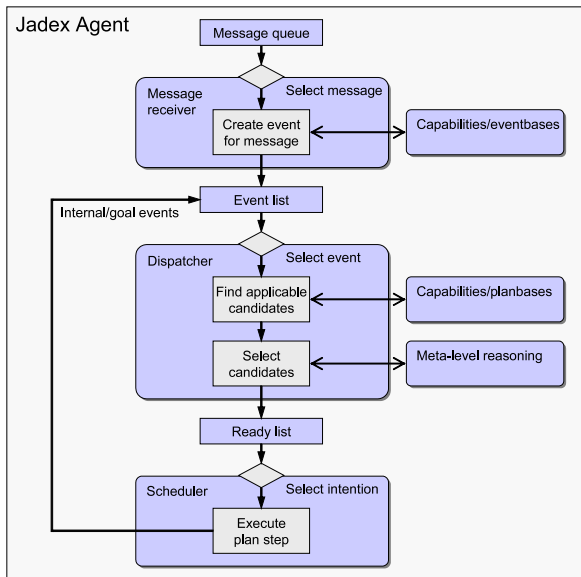
- **Plans** represent the behavioural elements of an agent and are composed of a **head and a body part**.
- The plan head specification is similar to other BDI systems and mainly specifies the circumstances under which a plan may be selected.
- In the **plan head** a **context condition** can be stated that must be true for the plan to continue executing.
- The **plan body** provides a **predefined course of action**, given in a procedural language.

Concepts within Jadex

Capabilities

- **Capabilities** represent a grouping mechanism for the elements of a BDI agent, such as beliefs, goals, plans, and events.
- Closely related elements can be put together into a reusable module, which encapsulates a certain functionality.
- The enclosing capability of an element represents its scope, and an element only has access to elements of the same scope.
- To connect different capabilities, flexible import / export mechanisms can be used that define the external interface of the capability.

Execution Model



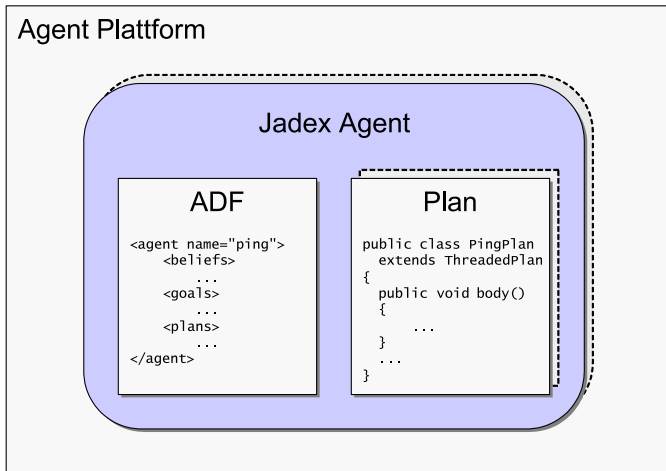


Figure 35: Jadex agent

Specifications and Syntactical Aspects

The **Jadex BDI metamodel** defined in **XML Schema** is very extensive. **Two design principles:**

1. Support for strong typing and explicit representation of all kinds of elements (beliefs, goals or events):
 - This requires users to write detailed **ADFs**, but in return allows for more rigorous consistency checking of agent models;
 - At runtime certain kinds of failures can be discovered more easily, e.g. the attempt of storing a fact value in an undefined belief can be immediately reported.

2. **Increasing the expressive power** of the **ADF** for the following purposes:

- The arbitrary complex creation of objects (e.g. values within beliefs or parameters);
- The description of boolean conditions (e.g. when a certain goal should be dropped);
- The construction of queries (e.g. for retrieving values from the beliefbase).

- An **embedded expression language** is used for **specifying parts of the agent model**, not easily represented in XML.
- **Expressions** are used throughout the **XML ADF**, whenever values have to be obtained for certain elements at runtime, e.g. values of beliefs, conditions of goals, etc.
- **Expressions** should be **side effect free**, because they are often evaluated internally by the system.
- The **expression language** has been designed to **fully comply with the syntax of Java expressions** (right hand side of assignments) **extended with a subset of OQL** (object query language) instructions.

```

01: select_expression ::= "SELECT" ("ALL" | "ANY" | "IOTA")?
02: (
03:   expression "FROM" ("$" identifier "IN" expression) ("," "$" identifier "IN" expression)*
04:   | "$" identifier "FROM" expression
05: )
06: ("WHERE" expression)?
07: ("ORDER" "BY" expression ("ASC" | "DESC")?)?

```

Example: **SELECT** \$block **FROM** \$beliefbase.blocks **WHERE** \$block.isClear()

Figure 36: OQL syntax in EBNF and query example

- It allows for query statements being created in the well-known **select-from-where** form, whereby:
 - It can be additionally specified if exactly one (**iota**);
 - The first satisfying (**any**);
 - Or **all** satisfying results are expected.

(Line 1)

- In the **from** clause (**lines 3–4**) it is specified from which object set (**line 4**) or joined sets (**line 3**) results are generated.
- The **identifiers define variables**, which iterate over the object sets specified as arbitrary expressions.
- These iterated values are checked against the boolean **where** condition (**line 6**) and can possibly be ordered (**line 7**).

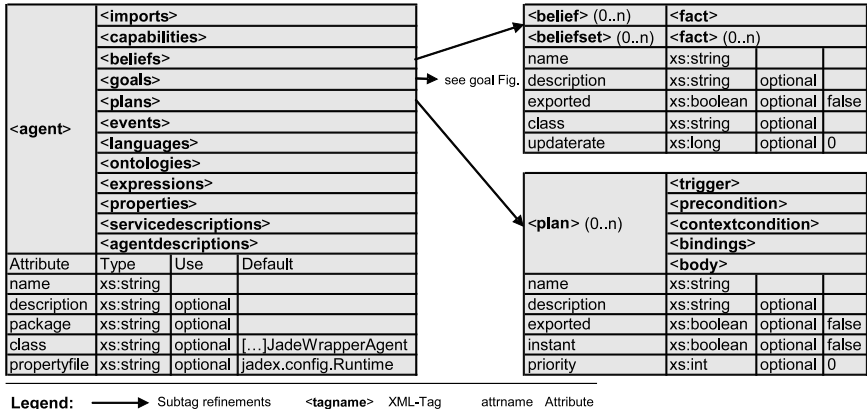


Figure 37: Agent metamodel specification fragment (XML-schema)

Specifications and Syntactical Aspects

Beliefs

- In **Jadex**, **beliefs** are represented in an **object-oriented way** allowing arbitrary Java objects being stored as facts.
- Like all elements of a **capability**, **beliefs and belief sets** can be supplied with a **name**, a **description text** and an **exported flag**.
- **Exporting** an element makes it **accessible from the outer scope** (respectively a capability or an agent) and is turned off by default.
- For beliefs and belief sets, the Java class for facts must be defined.

- Besides the type-relevant information, **initial fact data** can also be supplied **for configuring an agent's mental state at creation time**.
- The **value of a fact** has to be stated in the **expression language** and can be declared as **static or dynamic**.
- **Re-calculation** of such dynamic facts occurs on access and additionally in fixed time intervals (using the update rate).
- At runtime, **beliefs and belief sets** are **accessible** from within plans **via operations on the beliefbase** and additionally by issuing **OQL-like queries**.

Specifications and Syntactical Aspects

Goals

- Four different **goal types** are distinguished (**perform, achieve, maintain and query**).
- All these goal types are based on the **generic life cycle** and hence exhibit many common properties that are summarized in an **abstract base goal type**.
- **Creation, drop and context conditions** can be specified as **boolean expressions**.
- **Customization of goal types** can be further achieved by **defining named in-, out- and inout-parameters** that are used to **transfer information between a goal's originator and its processing plans**.

- **Binding parameters** can be used **for generating one goal instance** for every possible binding.
- The **runtime processing of goals** can be refined using the various BDI-flags, which inter alia control:
 - If a goal is retried when a plan fails (**retry**);
 - If meta-level reasoning is used (**mlreasoning**);
 - If applicable plans are tried sequentially or in parallel (**posttoall**).

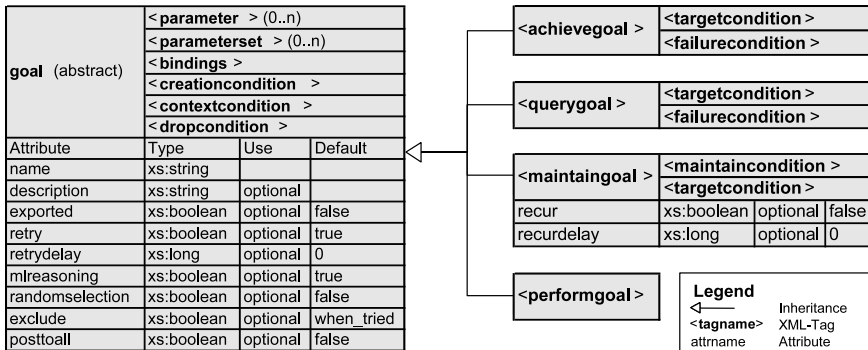


Figure 38: Goal metamodel specification (XML-schema)

Specifications and Syntactical Aspects

Plans

- The declaration of **plans** in **Jadex** is very similar to other **PRS-like systems** and **requires** the **specification of the plan heads** describing the circumstances under which a plan is applicable in the ADF.
- As plan trigger, internal events, messages, and goals, as well as a belief state condition (for data driven plans) can be provided.
- The **pre- and context condition** of a plan can be specified as **boolean expressions**.

- To facilitate **goal achievement** with plans, it is sometimes advantageous to create **several different parametrized plan instances** of a plan type and try them one after another until a plan succeeds.
- For this purpose, **binding parameters** can be specified and used **for plan configuration**.
- The **selection** of which plan is executed in response to an occurring trigger can be adjusted **by setting a priority value**.
- As **part of the initial mental state** of an agent, it can be further declared whether a plan is instantiated when the agent is created (using the **instant flag**).

- The **plan body** needs to be supplied as **expression for the creation of a suitable plan instance**.
- **Two different types of plan bodies** are supported, which both require a **Java** class to be implemented:
 - 1 **Standard plan bodies**;
 - 2 **Mobile plan bodies**.

```
01: /** Plan skeleton for an application plan. */
02: public class SomePlan extends jadex.runtime.Plan {
03:
04:     public void body() {
05:         // Plan code.
06:     }
07:
08:     public void passed() {
09:         // Optional cleanup code in case of a plan success.
10:     }
11:     public void failed() {
12:         // Optional cleanup code in case of a plan failure.
13:     }
14:     public void aborted() {
15:         // Optional cleanup code in case the plan is aborted.
16:     }
17: }
```

Figure 39: Plan skeleton

Software Engineering Issues

- The **overall goal** of the **Jadex** project is to **provide a sophisticated reasoning engine** allowing to **develop arbitrary complex intelligent agents**.
- Therefore, while trying to be **as easily useable as possible**, the **system does not sacrifice expressiveness for simplicity**.
- **Software engineering issues** play an **important role in the design of the system**.
- A **primary goal** of the project is to facilitate a **smooth transition from mainstream object-oriented software development to an agent-oriented approach**.
- This is achieved by resorting to **established techniques** wherever possible.
- Another advantage is that the developer can continue to operate in a familiar environment.

- The system provides **advanced software engineering features**, such as **reusability** and **consistency checking**.
- The **capability concept** allows **encapsulating agent functionality** into a reusable module while **maintaining the abstraction level of BDI elements**.
- The explicit specification and strong typing of beliefs, goals, etc. facilitates consistency checks of ADFs to detect errors (e.g. spelling mistakes) as early as possible.

Example

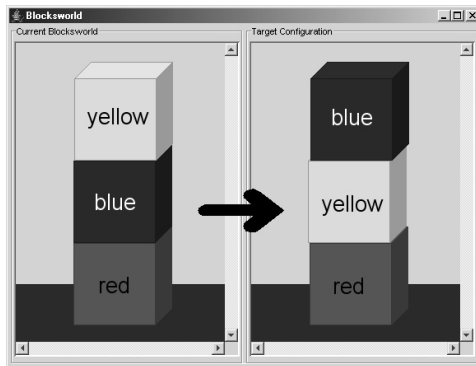


Figure 40: Blocksworld scenario

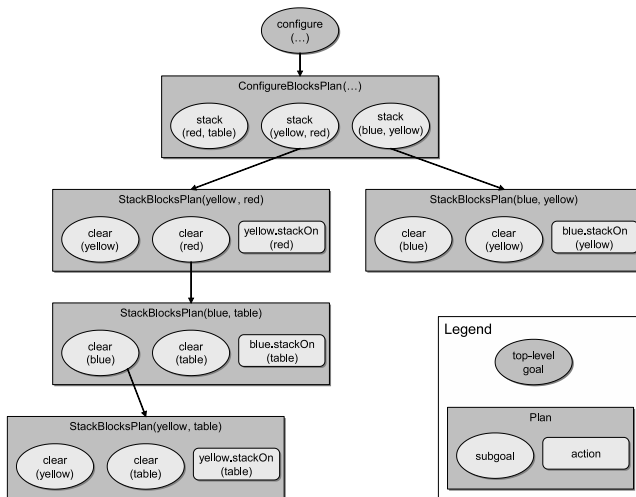


Figure 41: Goal/plan tree

Example

ADF of the agent

```
01: <agent name="Blocksworld" package="jadex.examples.blocksworld">
02:   <imports>
03:     <import>java.awt.Color</import>
04:   </imports>
05:
06:   <beliefs>
07:     <belief name="table" class="Table">
08:       <fact>new Table()</fact>
09:     </belief>
10:     <beliefset name="blocks" class="Block">
11:       <fact>new Block(new Color(240,16,16),$beliefbase.table)</fact>
12:       <fact>new Block(new Color(16,16,240),$beliefbase.table.allBlocks[0])</fact>
13:       <fact>new Block(new Color(240,240,16),$beliefbase.table.allBlocks[1])</fact>
14:       ...
15:     </beliefset>
16:   </beliefs>
17:
```

Figure 42: Blocksworld agent model

```
18: <goals>
19:   <achievegoal name="clear">
20:     <parameter name="block" class="Block" />
21:     <targetcondition>$goal.block.isClear()</targetcondition>
22:   </achievegoal>
23:   <achievegoal name="stack">
24:     <parameter name="block" class="Block" />
25:     <parameter name="target" class="Block" />
26:     <targetcondition>$goal.block.lower==$goal.target</targetcondition>
27:   </achievegoal>
28:   <achievegoal name="configure">
29:     <parameter name="configuration" class="Table" />
30:     <targetcondition>
31:       $beliefbase.table.configurationEquals($goal.configuration)
32:     </targetcondition>
33:   </achievegoal>
34: </goals>
35:
```

Figure 43: Blocksworld agent model

```
36: <plans>
37:   <plan name="stack">
38:     <body>new StackBlocksPlan($event.goal.block, $event.goal.target)</body>
39:     <trigger><goal ref="stack"/></trigger>
40:   </plan>
41:   <plan name="configure">
42:     <body>new ConfigureBlocksPlan($event.goal.configuration)</body>
43:     <trigger><goal ref="configure"/></trigger>
44:   </plan>
45:   <plan name="clear">
46:     <bindings>
47:       <binding name="upper">
48:         select $upper from $beliefbase.blocks where $upper.lower==$event.goal.block
49:       </binding>
50:     </bindings>
51:     <body>new StackBlocksPlan($upper, $beliefbase.table)</body>
52:     <trigger><goal ref="clear"/></trigger>
53:   </plan>
54: </plans>
55: </agent>
```

Figure 44: Blocksworld agent model

Example

Java code for StackBlocksPlan

```
01: package jadex.examples.blocksworld;
02: import jadex.runtime.*;
03:
04: /** Plan to stack one block on top of another target block. */
05: public class StackBlocksPlan extends Plan {
06:     protected Block block;
07:     protected Block target;
08:
09:     public StackBlocksPlan(Block block, Block target) {
10:         this.block = block;
11:         this.target = target;
12:     }
13:
```

Figure 45: Java code for StackBlocksPlan


```
14: public void body() {  
15:   IGoal clear = createGoal("clear");  
16:   clear.getParameter("block").setValue(block);  
17:   dispatchSubgoalAndWait(clear);  
18:  
19:   clear = createGoal("clear");  
20:   clear.getParameter("block").setValue(target);  
21:   dispatchSubgoalAndWait(clear);  
22:  
23:   block.stackOn(target);  
24: }  
25: }
```

Figure 46: Java code for StackBlocksPlan

Example

Java code for ConfigureBlocksPlan

```
01: package jadex.examples.blocksworld;
02: import jadex.runtime.*;
03:
04: /** Plan to establish a given configuration of blocks. */
05: public class ConfigureBlocksPlan extends Plan {
06:     protected Table table;
07:
08:     public ConfigureBlocksPlan(Table table) {
09:         this.table = table;
10:     }
```

Figure 47: Java code for ConfigureBlocksPlan

```
11:
12: public void body() {
13:     Block[][] stacks = table.getStacks();
14:     for(int i=0; i<stacks.length; i++) {
15:         for(int j=0; j<stacks[i].length; j++) {
16:             Block block=(Block)getBeliefbase().getBeliefSet("blocks").getFact(stacks[i][j]);
17:             Block target=stacks[i][j].getLower()==table
18:             ?(Table)getBeliefbase().getBelief("table").getFact()
19:             :(Block)getBeliefbase().getBeliefSet("blocks").getFact(stacks[i][j].getLower());
20:
21:             IGoal stack = createGoal("stack");
22:             stack.getParameter("block").setValue(block);
23:             stack.getParameter("target").setValue(target);
24:             dispatchSubgoalAndWait(stack);
25:         }
26:     }
27: }
28: }
```

Figure 48: Java code for ConfigureBlocksPlan

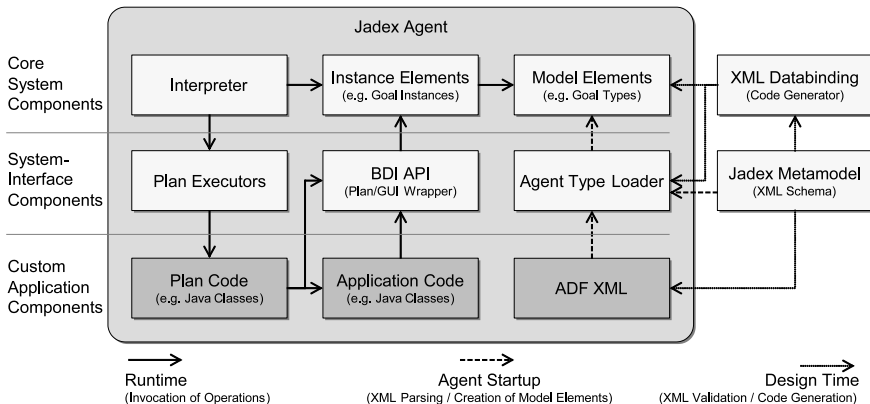


Figure 49: System realization

Available tools and documentation

- The system distribution contains **complete documentation materials for quick start and reference purposes**.
- An **introductory tutorial** made up of several exercises shows the usage of basic system features in a step-by-step manner.
- The distribution provides **several example applications** including their commented source code.
- A **user guide** provides a systematic overview of all features and also serves as a reference manual.
- **Javadocs** of the plan programming API and a reference to the metamodel defined in XML Schema are provided.

- The **available tools** are covered in a separate guide.
- There are publicly available online tools kindly hosted by **SourceForge.net**, such as:
 - **Web forums for discussion and support requests;**
 - **A database for bug-reports and feature requests;**
 - **A general mailing list with online archives.**

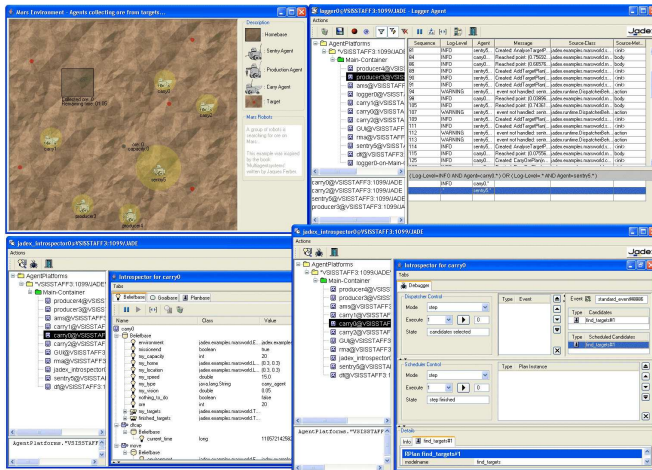


Figure 50: BDI introspector and logger screenshots

- **All runtime tools** provided by the **JADE** platform such as **Sniffer and Dummy agent** can also be used with **Jadex** agents.
- To enable a **comfortable testing** of the internals of **Jadex** agents **additional tool agents** have been developed.
- In Fig. ?? an example application (marsworld) is depicted together with the logger and introspector tools in a typical debugging session.

- The **BDI introspector** (Fig.?? bottom left and right hand side) **serves two purposes**:
 - It **supports the visualization and modification of the internal BDI concepts** thus **allowing inspection and reconfiguration** of an agent at runtime.
 - It **simplifies debugging** through a facility for the stepwise agent execution.
- In the **step mode**, it is possible to **observe and control** each **event processing and plan execution step** having detailed **control over** the **dispatcher and scheduler**.
- It can be easily figured out what plans are selected for a given event or goal.

- With the help of the **logger** (see Fig.?? on the top right) the **agent's outputs** can be directed to a single point of responsibility at runtime.
- In contrast to simple console outputs, the **logger agent** preserves **additional information** about the output such as its time stamp and its source (the agent and method).
- Using these artifacts the **logger agent** offers **facilities for filtering and sorting messages** by various criteria allowing a personalized view to be created.

- A **tracer tool** for **on-line visualization of agent execution** based on ideas from is provided.
- It generates a **unified view of multi-agent and internal agent behaviour**, relating message-based communication and internal agent processes.
- The **Jadexdoc tool** allows generating documentation of agent applications similar to Javadoc.
- In addition to these tools already included in the latest release, a **tool for multi-agent application deployment** is currently in development.

Standards compliance, interoperability and portability

- One driving factor for the development of **Jadex** was the need for a **FIPA-compliant platform** supporting **advanced BDI reasoning capabilities**.
- **FIPA-compliance** is achieved through the **JADE** platform, which provides sophisticated implementations of all important **FIPA specifications**.
- The **Jadex** reasoning engine, realized on top of the **JADE** platform, in itself only supports homogeneous (i.e. BDI) agents, but provides interoperability with agents based on other models.

- Agents realized using the conventional **JADE** programming techniques can be executed directly together with **Jadex** agents running on the same platform.
- Interoperability with other kinds of agents is straightforward as long as those agents adhere to the **FIPA standard**.

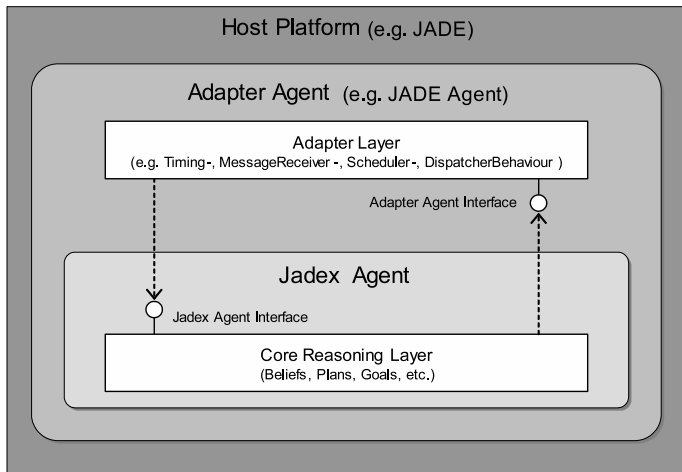


Figure 51: Platform integration

- The reasoning engine has been realized as a separate component, intentionally limiting the dependencies to the underlying platform.
- To use the reasoning engine on top of other platforms, an adapter has to be realized.
- This adapter has to implement a handful of methods used by the **Jadex** engine (e.g. to send messages) and has to call the engine when it is expected to do the reasoning.

- Therefore, although the current implementation is designed to be used with **JADE** , the reasoning engine can be easily integrated with other FIPA-compliant agent platforms such as CAPA or ADK, given that they provide a similar interface for message handling.
- It is also possible to use the system in conjunction with other **middleware** environments such as **J2EE** or .NET, when FIPA-compliance is not needed.
- Currently, in addition to the JADE integration, we have developed experimental adapters for the DIET agent platform and for running a set of Jadex agents as a standalone Java application.

- The engine was realized in Java 1.4 and includes the third party packages JBind for XML data binding and Apache Velocity for generating the content of some tool dialogs.
- To support mobile devices, a port of the engine is also available in a reduced version based on **J2ME** / CDC.
- Moreover, all kinds of tools and libraries with a Java API can easily be used to provide additional features.
- For example, in a larger project the Cayenne database mapping framework was used to connect agents to a relational database.

- **Jadex** is a general-purpose development environment for creating multi-agent system applications, allowing to build agents with reactive (event-based) and deliberative (goal-driven) behaviour.
- It is not bound to a specific target domain, but has been used to realize applications in different domains such as simulation, scheduling, and mobile computation.
- **Jadex** originated in the MedPAGe (“Medical Path Agents”) project, which is part of the German priority research programme 1083 **Intelligent Agents in Real-World Business Applications** funded by the Deutsche Forschungsgemeinschaft (DFG).
- In cooperation with the business management department of the University of Mannheim, the project investigates the advantages of using agent technology in the context of hospital logistics.

- In this project Jadex is used to realize a multi-agent application for market-based negotiation of treatment schedules, as well as for the simulation of a hospital model to test the negotiation mechanism.
- In other contexts, **Jadex** was used to realize portable PDA-based applications.
- A personal mobile task planner was developed, to test the **Jadex** J2ME port and to prove the usefulness of BDI agents on mobile devices.
- Elsewhere, in the PITA (“Personal Intelligent Travel Assistant”) project at the Delft University of Technology, **Jadex** was used to realize a prototype of a mobile personal travel assistant application

- Besides building specific agent applications, **Jadex** has also been used for teaching and research regarding agent oriented software development in general.
- Due to its simple language based on well-known technologies such as Java and XML, and the extensive documentation material and illustrative example applications, **Jadex** is well suited for teaching purposes.
- It has been successfully applied in several courses at the University of Hamburg, and is also evaluated by other institutes.
- Regarding research in agent systems, the project is also designed as a means for researchers to further investigate which mentalistic concepts are appropriate in the design and implementation of agent systems.

- The combination of XML Schema with Java databinding techniques allows the **Jadex** metamodel to be flexibly adapted and extended for experimentation purposes.
- While investigating different representations for beliefs, goals and plans, the system has been applied to several well-known AI problem domains (blocksworld, cleanerworld, mars robots, hunter-prey).
- These applications are also included in the distribution.
- Moreover, the Technical University of Karlsruhe has used **Jadex** to implement an experimental system for representing norms in multi-agent systems.

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** Reactive and deliberative behaviour is supported based on the BDI model and the corresponding mental attitudes. In addition to the basic BDI interpreter known from PRS systems, an explicit representation of goals is provided.

Communication

Q: Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?

A: FIPA-compliant speech-act based communication is provided by the underlying **JADE** platform.

Underlying Computational Model

- Q:** Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?
- A:** Weak mobility is provided by the underlying **JADE** platform. When developing mobile agents, some features of the system (e.g. thread-based plans) are not available.

Simplicity

Q: How easy it is to use and understand the language?

A: The language is easy to learn, as it is based on well-known technologies such as Java and XML. Experiments with students have shown that new users are quickly able to develop their first agents.

Preciseness

Q: Does the language have clear and precise semantics? How has it been formalised?

A: No formal semantics is available.

Expressiveness

Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?

A: The language is very general and allows creating different kinds of agent applications.

Extensiveness

- Q:** Does the language allow the definition of new language components from the basic constructs in a systematic way?
- A:** The system does not define a new language for programming agent behaviour, but instead makes BDI-specific agent facilities available as application program interface (API). Hence, the BDI feature set can be easily extended.

Verification

Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?

A: No path to formal verification is provided.

Software Engineering Principles

- Q:** Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?
- A:** The XML language enforces strong typing. The plan language inherits the software engineering and programming language principles of Java. In addition, reusability is supported by the definition of agent-modules called capabilities.

Language Integration

- Q:** Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?
- A:** Embedding the agent language into a general-purpose language is not necessary, because the system cleanly separates the definition of an agent's structure and the definition of agent behaviour. The structure of an agent is defined in a system specific XML dialect following a BDI-metamodel, while the agent behavior is realized as plans coded directly in the general-purpose programming language Java.

Language Integration (continued)

- Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?
- A: The default plan language is Java and therefore allows accessing any other application code or third party library written in Java. In addition, it is possible to define wrappers that allow executing plans written in other (e.g. visual) languages.

Platform

Deployment and Portability

- Q:** Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A:** The documentation includes an introductory tutorial, a user guide, which also serves as a reference manual, and a guide to the available tools. Javadocs of the plan programming API, and a reference to the metamodel defined in XML Schema are provided, and the distribution includes several example applications with source.

Deployment and Portability (continued)

- Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?
- A: The system is based on Java 1.4, and requires a host agent platform such as **JADE** (which is currently supported best). The distribution includes the third party packages JBind for XML databinding, Apache Velocity for generating the content of some tool dialogs, and the TouchGraph GraphLayout component for visualizing traces of agent execution.

Standards Compliance

Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?

A: The system complies with the FIPA-standards as implemented by **JADE**

.

Platform Extensibility

- Q:** Can the platform be extended with additional functionality, for example through Open Source collaboration?
- A:** The system is Open Source and carefully designed and documented to allow easy and flexible extension of the provided functionality.

Available Tools

- Q:** What tools are provided by the platform for the management, monitoring, logging and debugging of applications?
- A:** In addition to the tools provided by the **JADE** platform such as Sniffer and Dummy Agent, the system supplies an introspector tool to inspect an agent at runtime, and to execute agents step-by-step. A logger agent allows to collect, filter, and view logging outputs and a tracer agent visualizes event traces produced by the different agents of a multi-agent application. The Jadexdoc tool generates documentation for an agent application.

Available Tools (continued)

Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?

A: Apart from the documentation material included in the distribution (e.g. user guide and tutorial), there are publicly available web forums for discussion and support requests, a database for bug-reports and feature requests, and a general mailing list with online archives.

Available Tools (continued)

- Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?
- A: No additional tools (apart from those provided by **JADE**) are yet available, but a tool for multi-agent system deployment is currently in development.

Tool Integration

- Q:** In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?
- A:** All kinds of tools and libraries with a Java API can be used within **Jadex** . For example, in a larger project the Cayenne database-mapping framework was used to connect agents to a relational database.

Technical Interoperability

- Q:** Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?
- A:** Although its current implementation is targeted to run on top of **JADE** , the reasoning engine provides a general integration mechanism, and is designed to be used on top of any existing middleware. Therefore, it can be easily ported to other FIPA-compliant agent platforms such as CAPA or ADK and to other middleware environments such as J2EE or .NET.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: The performance of the system regarding the number of agents and messages is bounded by the performance of the underlying platform (e.g. **JADE**). The computation cost induced by the reasoning engine highly depends on the complexity of the agents.

Performance Issues (continued)

- Q:** What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?
- A:** The system is available as stable Open Source distribution and has already been used in several 3rd party projects. Nevertheless, the set of features is continuously evolving, and compatibility between releases is not guaranteed.

Multi-Agent Systems Features

- Q:** Does the platform support open multi-agent systems and heterogeneous agents?
- A:** The system realizes a specific internal agent architecture, and therefore itself does not address heterogeneity, but it is possible to run **Jadex** agents on the same platform as other **JADE** agents. Openness is supported in principle through FIPA-compliant communication, but not especially facilitated by the design of the system.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: **Jadex** agents use the distributed or centralized control structures provided by the underlying platform (e.g. **JADE**). A hierarchical structure of agents is not supported, but agents can be decomposed into hierarchically structured modules, which are similar to agents, but do not have their own reasoning process.

Multi-Agent Systems Features (continued)

- Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?
- A: The system includes a ready-to-use module for communication with a directory facilitator (DF) and for using simple FIPA interaction protocols (e.g. request).

Applications Supported by the Language and Platform

Typical Examples

- Q:** What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?
- A:** The system has been used mainly in research projects and teaching courses, e.g. to realize a multi-agent application for market-based negotiation of patient treatment dates, as well as for the simulation of a hospital model. The system has also been applied in mobile environments and to some well-known AI problem domains such as blocksworld and cleanerworld.

Targeted Domains

Q: Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?

A: The system is general purpose and not bound to a particular application domain.

References I



M. Beelen.

Personal Intelligent Travelling Assistant: a distributed approach.

Master of science thesis, Knowledge Based Systems group, Delft University of Technology, 2004.



L. Braubach, A. Pokahr, K.-H. Krempels, and W. Lamersdorf.

Deployment of Distributed Multi-Agent Systems.

In *Fifth International Workshop on Engineering Societies in the Agents World (ESAW 2004)*, 2004.



L. Braubach, A. Pokahr, W. Lamersdorf, K.-H. Krempels, and P.-O. Woelk.

A Generic Simulation Service for Distributed Multi-Agent Systems.

In *From Agent Theory to Agent Implementation (AT2AI'04)*, 2004.

References II



L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf.

Goal Representation for BDI Agent Systems.

In *Proceedings of the Second Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04)*, 2004.



P. Busetta, N. Howden, R. Rönquist, and A. Hodgson.

Structuring BDI Agents in Functional Clusters.

In N. R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL) '99*, pages 277–289. Springer, 2000.

References III



M. Duvigneau, D. Moldt, and H. Rölke.

Concurrent Architecture for a Multi-agent Platform.

In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III. Third International Workshop, AOSE 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*, pages 59–72.

Springer, 2003.



E. Norling.

Folk Psychology for Human Modelling: Extending the BDI Paradigm.

In *Proceedings of in the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, Ney York, USA, July 2004.

References IV



T. O. Paulussen, N. R. Jennings, K. S. Decker, and A. Heinzl.

Distributed Patient Scheduling in Hospitals.

In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann, 2003.



T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, and W. Lamersdorf.

Dynamic Patient Scheduling in Hospitals.

In M. Bichler, C. Holtmann, S. Kirn, J. Mä $\frac{1}{4}$ ller, and C. Weinhardt, editors, *Coordination and Agent Technology in Value Networks*. GITO, Berlin, 2004.



A. Pokahr, L. Braubach, and W. Lamersdorf.

Jadex: Implementing a BDI-Infrastructure for JADE Agents.

EXP – in search of innovation, 3(3):76–85, 2003.

Abstract

Software agents offer a range of benefits to the development of complex software systems. However, before these benefits can be realised by the computing industry there is a need for an agent platform that can be accepted by industry. In this paper we describe the **JACK** agent platform: a mature and robust commercial product. We argue that **JACK** meets requirements such as familiarity, scalability and integrability which make it suitable for adoption by industry. We also describe interesting features of **JACK** such as the use of capabilities for structuring agents, and **JACK** 's approach to teamwork which allows hierarchical team structures.

Keywords

Agent Oriented Programming Language, Belief-Desire-Intention, Agent Platform.

Author

Michael Winikoff

- **Software agents** offer a **range of potential benefits** to the development and deployment of complex software systems:
 - Increased **flexibility** and **adaptability**;
 - More **natural models** of complex “nearly decomposable” systems.
- These benefits stem from the combination of features that are generally considered to be associated with **intelligent software agents**:
 - **Autonomous**;
 - **Proactive**;
 - **Reactive**;
 - **Social**.

- **Some argue** that because **agents** are autonomous they **reduce coupling**.
- **Some** focus on the use of **plans** and **goals** in **Belief-Desire-Intention (BDI)** agents (and similar platforms), **arguing** that the resulting number of ways in which a goal can be achieved gives agents **flexibility** in dealing with situations, and **robustness** in recovering from various types of failure.
- **Others argue** that **aspects of agents** (e.g., autonomy, flexibility) are **already being adopted by mainstream software engineering**, and that this is evidence that these aspects are useful to modern software systems.

- In order for these benefits to be realisable by the computing industry, a number of **key technological pieces** are required.
- One of the key pieces is a **methodology** (including concepts, notations, a process and techniques) that guides practitioners in designing agent systems.
- A number of methodologies have been developed:
 - **Gaia**;
 - **Tropos**;
 - **Prometheus**;
 - **MaSE**;
 - **ROADMAP**.

A second key piece of technology is an **agent platform** which can be used to create agent systems.

We believe that an agent platform needs to contain at least the following components:

- An **agent-oriented programming language** that allows agents to be written directly using agent concepts (e.g., plans, goals, beliefs), rather than encoded in non-agent-oriented languages.
- A **library or framework** providing facilities for **inter-agent communication** including facilities for transmitting and receiving messages, and for **locating agents** (e.g., a name server).

- **JACKTMIntelligent Agents (JACK)** is an agent platform that includes these components and more.
- **JACK** includes:
 - An **agent-oriented programming language**;
 - A **platform** for executing agents **with infrastructure** such as message marshalling and a name server;
 - **Development tools** including a **design tool**, a **graphical plan editor** and a number of **debugging views**;
 - A **number of additional functionalities** such as the ability to construct hierarchical **teams** of agents.

We believe that to be **acceptable to industry** an **agent platform must be:**

- Familiar;
- Integratable;
- Scalable;
- Industrial Strength (Robust, Stable, Efficient);
- Documented and Supported.

Additionally, it is important for the agent platform to **provide development tools** and **debugging tools**.

- The JACK programming language **extends Java** in a number of ways, both syntactic and semantic.
- The JACK language is a **superset of the Java programming language**, so all of Java's libraries and facilities are easily accessible.

Specifications and Syntactical Aspects

Syntactically, **JACK** extends Java in three ways:

- ❶ **JACK** adds new top-level declaration types which are used to declare agents, beliefsets, views, events, plans and capabilities.
- ❷ Each of the top-level types is defined using various # declarations which define the properties of the entity and relationships between entities.
- ❸ Within plan bodies **JACK** defines a range of @ statements such as:
 - Posting an event (e.g., @post);
 - Waiting for a condition (@wait_for);
 - Some of the @ statements defined by JACK are listed in figure ??.

- `@post`, `@subtask` – simple event posting within an agent. `@post` is asynchronous, whereas `@subtask` waits for the event processing to finish before continuing.
- `@send`, `@reply` – inter-agent communication.
- `@achieve`, `@insist` – post a (goal) event under certain conditions.
`@achieve(condition, goal_event)` checks whether the condition holds, and posts the event if it doesn't. `@insist(condition, goal_event)` is similar, but also checks whether the condition holds after the processing triggered by the event has finished. If not, the event is posted again.
- `@maintain` – checks for condition while handling event.
`@maintain(condition, event)` will subtask the event, but will monitor the condition while the event processing runs. If the condition becomes false the plans that handled the event are aborted.
- `@sleep`, `@wait_for` – do nothing for a certain amount of time (`@sleep`) or until a certain condition is true (`@wait_for`).

Figure 52: Some statements provided by JACK

```
public plan ProcessRequest extends Plan {  
    #handles event Request req;  
    #sends event Response resp;  
  
    context() {  
        req.isValid;  
    }  
  
    #reasoning method body() {  
        // Can contain Java code as well  
        // as JACK @-statements  
        ...  
        @reply(req, resp.response(...));  
    }  
}
```

Figure 53: A (very simple) Plan

Figure ?? shows how these three syntactic extensions are used to define a (very simple) plan called `ProcessRequest`:

- which is triggered by a message (declared with the `#handles` declaration);
- and replies to it with a response.

The **top-level entities** that **JACK** defines are:

Agent: Obvious basic entity for an agent-oriented programming language! Specified by:

- Events they handle and send;
- Data (including beliefsets) they have;
- Plans and capabilities they use.

Beliefset: A beliefset is effectively a (small) relational database that is stored in memory, rather than on disk.

View: Views are “virtual” beliefsets that are computed from other beliefsets.

The **top-level entities** that **JACK** defines are:

Event: An event is an occurrence in time that represents some sort of change that requires a response.

Plan: A plan is a “recipe” for dealing with a given event type. Plans include

- An indication of which event they handle;
- A **context condition** which describes in which situations the plan can be used;
- A plan body.

Capability: A capability is a modularisation construct.

- ❶ Event posted.
- ❷ Determine the set of **relevant plans**.
- ❸ Determine the **applicable plans**.
- ❹ Select an applicable plan and run it.
- ❺ If plan fails, go to step 4 (select an applicable plan).

Figure 54: Event handling in BDI architectures

- The execution of a plan's body is fairly straightforward: the statements in the plan body are executed in sequence.
- However, there is one key difference between executing Java code and executing **JACK** code: each statement can fail, and if it does the rest of the plan is not executed and failure handling is triggered instead.
- When a plan fails the event that triggered it is considered to have not been handled, and alternative plans for handling it are considered.
- This process looks for another applicable plan to try.
- If there is another applicable plan, it is tried.
- If all applicable plans have failed the event cannot be handled.
- If the event was posted from a plan that plan fails and its triggering event is re-posted in an attempt to find an alternative applicable plan for it.

- This execution cycle of events triggering plans is common to a whole family of BDI architectures.
- However, there are some details of the cycle that are specific to JACK and distinguish it from other platforms.
- By default **JACK** re-computes the applicable set when considering alternative plans due to failure.
- This means that when a plan fails and alternatives are considered, the applicability of these alternatives is evaluated in the current situation, not the situation when the event was first posted.
- Some other BDI architectures (such as JAM) do not re-compute the applicable plan set, and thus select plans based on out-of-date information when failure occurs.

- Another detail that is specific to JACK is that the context condition is actually split into two parts: a **context condition** and a **relevance condition**.
- The relevance condition is a Boolean condition that is only evaluated once (eagerly) and can only access the details of the event, not any other data.
- The relevance condition is used to exclude plans based on the details of the event (which do not change).
- For example, if the event is a request for credit which specifies the amount and there are separate plans depending on the amount requested, the selection of plans can be done using a relevance condition rather than a context condition.

- When there are multiple applicable plans that can be used, the question arises of which one an agent should select (step 5 in figure ??).
- **JACK** provides a number of mechanisms that allow the programmer to specify how a plan should be selected.
- One mechanism is that plans will (by default) be selected in the order in which they are listed in the agent.
- Another mechanism runs another plan (a “meta-plan”) to decide which plan to select.
- **JACK** actually provides a variety of event types which behave differently.
- For example, message events do not trigger failure handling if their handling plan fails.

- ❶ Event posted.
- ❷ Find plans that `#handle` it.
- ❸ Determine the set of relevant plans using `relevant()` method (Relevant method can only access event, not beliefs).
- ❹ Determine applicable plans using `context()`.
- ❺ Select a plan and run its `body()`. (meta-reasoning can be used to make the selection).
- ❻ If plan fails, go to step 4 (recompute applicable plan set).

Figure 55: Event handling in JACK

Semantics and Verification

- Although **JACK** is quite well documented, its **semantics** have **not been formally specified**.
- Since **JACK** is a **superset of Java**, formally defining **JACK** 's semantics would require a formal definition of Java's semantics, something that is still an active area of research.
- The **event-plan execution cycle** which **JACK** shares with other BDI platforms has been **formalised** in various ways by various researchers.

- Anand Rao's work on **AgentSpeak(L)** aims to bridge the “BDI gap” between theories and implementations by defining a language capturing the essence of BDI platforms whilst having precisely defined semantics.
- Although the formal semantics given by Rao is incomplete, the work has inspired a number of implementations of the language such as:
 - **AgentTalk**, an implementation based on SIM_AGENT;
 - An implementation in Java that is designed to run on hand-held devices;
 - The Java-based **Jason**.

- Since Rao introduced **AgentSpeak(L)**, a number of authors have published **complete formal semantics** for the language.
- The specification language Z (“Zed”) was used to formally specify the essential execution cycle of AgentSpeak, and an operational semantics for AgentSpeak was given by Bordini and Moreira.
- However, neither of these formalisations included the failure handling mechanism.
- A precise operational semantics including failure handling was given by Winikoff *et al.* for a language (called “**CAN**”) which is a **superset of AgentSpeak**.

- Since JACK's semantics has not been formalised, JACK programs cannot be formally verified.
- Verification of entire implemented systems is not currently realistic.
- Research into model checking of agent programs is still quite young, and is not yet applicable to large agent programs.
- We believe that presently formal techniques are best applied to verifying **aspects** of systems, such as key algorithms or interaction patterns.

Software Engineering Issues

- One of JACK's strengths is its **support for modern software engineering practices**.
- In addition to the features provided by Java (objects, packages), JACK adds a **number of features** that can be used to structure an agent system.
- One new feature is that a plan's body can be broken down into a number of separate **reasoning methods**, rather than being a single monolithic block of JACK code.
- This allows a single plan to be structured internally.
- Another feature that was introduced by JACK (and subsequently adopted by **Jadex**) is **capabilities**.

- A **capability** is the **agent-oriented equivalent of a module**, corresponding to a coherent ability that an agent has.
- **Capabilities** contain **plans and beliefs**, and specify which **events** they handle and post.
- **Capabilities** can also contain **sub-capabilities** which allow **hierarchical module structures** to be specified as appropriate.

- Another Software Engineering practice is **consistency checking**.
- **JACK** checks that the various declarations of which events are posted and handled by which entities (agents, capabilities and plans) are consistent.
- The JACK agent programming language is, as an extension of Java, strongly and statically typed, and the type checking done at compile-time can catch a range of mistakes made by the programmer.

Other features of the language

- JACK's support for **teams** is an optional extension which adds two new concepts (**teams** and **roles**) and extends Plans to **TeamPlans**.
- A team is an entity which, like an agent, can contain plans, capabilities, data, etc.
- Unlike agents, a team can also have **sub-teams**, enabling natural modelling of hierarchical organisational structures.
- A team is an active entity that can have beliefs and execute (team) plans; it is not merely a collection of agents.
- When the team extension is enabled, an individual agent is modelled simply as a team that has no sub-teams!
- For each team type, roles are used to specify the interface (in terms of events received and sent) that must be fulfilled by its sub-teams.

- The team extension also extends Plans to TeamPlans by adding the ability to delegate tasks to sub-teams, and to perform steps in parallel.
- Team plans also differ from plans in that they have an `establish()` reasoning method which assembles the sub-teams that will be involved in the plan (the “task team”).
- Each TeamPlan that is run by a team can have a different assignment of sub-teams.
- For example, given a team of soccer-playing robots, one TeamPlan may require two attackers, whereas another TeamPlan may require both a defender and a goal keeper.

```
teamplan FeedBaby extends TeamPlan {  
    #handles event BabyHungry pfv;  
    #uses role Parent parents as p1;  
    #uses role Parent parents as p2;  
  
    //establish the task team.  
    #reasoning method establish() { ... }  
  
    body() {  
        @parallel(ParallelFSM.ALL,false,null) {  
            @team_achieve(p1, p1.prepareFood.pf());  
            @team_achieve(p2, p2.calmBaby.cb());  
        };  
        @team_achieve(p2,p2,feedBaby.fb());  
    } // body  
} // FeedBaby team plan
```

Figure 56: A simple TeamPlan

- Another feature of JACK is an event type called `InferenceGoal`.
- This behaviour is useful for performing certain types of reasoning such as emulating rule firing in expert systems.
- The JACK compiler is modular.
- The JACK language can be extended using plugins.

Available tools and documentation

- The areas that were seen as most desirable to be supported by third party tools were:
 - “Integrated Development Environments;
 - Debugging tools;
 - Parsers/language tools”.
- **JACK** addresses the first two areas by providing an integrated development environment, and a range of debugging tools.

- The **JACK Development Environment** (JDE) allows the developer to create agents, events, plans, beliefsets etc. by dragging and dropping, rather than typing # declarations.
- The JACK skeleton code for the entities is automatically generated.
- The JDE also provides a **Graphical Plan Editor** which allows the bodies of plans to be specified using a graphical notation, rather than textual code.

- The JDE also includes a **Design Tool** which allows overview diagrams in the style of Prometheus to be drawn.
- This can be used to create the system's structure by placing entities onto the canvas and linking them together.
- It can also be used to create an overview of an existing system by adding entities to a canvas, in which case the links between entities are automatically added.
- The JDE maintains consistency between the design diagrams and the underlying model, and therefore with the generated code.

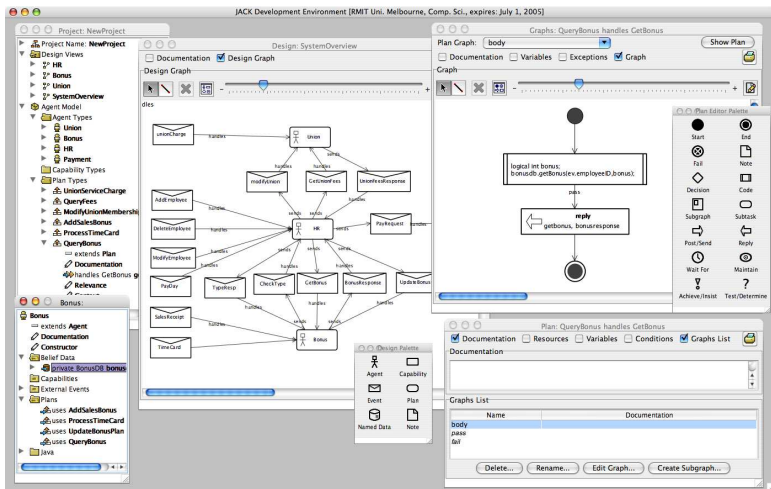


Figure 57: The JACK Development Environment (JDE)

JACK provides a number of **debugging tools**:

1. The simplest is a **textual trace** of processing steps which is enabled from the command line.
 - Can be configured to show various types of steps:
 - Changes to beliefsets;
 - Events being posted and processed;
 - Messages being sent and received;
 - Steps in plans.
 - Less useful for debugging distributed systems of agents.

JACK provides a number of **debugging tools**:

2. For debugging distributed agents **interaction diagrams** are more useful.
 - An interaction diagram graphically displays messages sent between agents.
 - A single interaction diagram can collect and display messages from agents across a distributed system.
 - Interaction diagrams depict the messages between agents.

JACK provides a number of **debugging tools**:

3. JACK provides **graphical plan tracing**, which traces the execution of plans that have been specified using the Graphical Plan Editor.
 - When a plan begins executing its graph is shown, and as the plan executes the currently executing node is highlighted.
 - The graph also shows the values of the plan's variables and parameters.
 - The execution of the agent can be controlled:
 - It can be run as normal;
 - Single-stepped;
 - Stepped with a delay in between steps.

The newest version of JACK also provides an **additional debugging tool**:

- A **browser** that allows the state of agents (including their beliefs and active tasks) to be inspected.

- All of these development, design and debugging tools – as well as the JACK language, and other facilities such as JACK's support for teamwork, the Webbot interface to JSP, and JACOB (see next section) – have clear and extensive documentation.
- JACK's documentation package also includes “practicals”: a tutorial sequence introducing JACK.

Standards compliance, interoperability and portability

- There are many approaches to communication and integration, such as CORBA, HLA, Java RMI and FIPA.
- JACK's approach to communications is agnostic.
- It is also possible to extend and/or replace JACK's communications infrastructure.
- We begin by discussing JACK's lightweight communications infrastructure including a discussion of **JACOB** .

- JACK's lightweight communications mechanism supports sending messages between agents.
- These messages can contain Java objects which are serialised by the sender and "reconstituted" by the recipient of the message.
- JACK provides a number of mechanisms for serialising objects: Java's serialisation can be used, but this tends to produce large messages, and only supports communication with other Java software.

- Alternatively, **JACOB** provides more compact serialisations, and allows objects to be “reconstituted” by Java or C++ programs.
- **JACOB** provides a number of serialisation formats:
 - A plain ASCII format that is compact yet human readable;
 - A binary format which is more compact;
 - An XML format;
 - A **JDBC** format.

- When the recipient of a message is in the same Java process as the sender, then the message is addressed simply using the name of recipient agent.
- **JACK** supports flexible distribution of agent. It is possible:
 - To have multiple agents per Java process;
 - To have agents distributed in different Java processes;
 - Or to flexibly mix these.
- This flexible distribution requires a slightly more sophisticated addressing scheme than simply using agent names, and JACK introduces the concept of a **portal**.
- A portal can be thought of as a handle on a Java process, and sending a message to an agent at another portal is done by addressing the agent as *agentname@portalname*.
- Each portal acts as a name server for other portals, i.e., each portal keeps track of the addresses of other portals.

Other features of the platform

JACK is efficient. It allows:

- Flexible distribution of agents, with multiple agents sharing a Java process.
- For many agents to run on a single machine, while still supporting distributed agent systems across machines.

- Benchmarking on an average PC running Linux shows that over 1000 agents can be created per second, and that 100,000 messages can be sent per second (within the same Java process).
- JACK is compact enough to be run on limited hardware.
- It has been demonstrated on a Psion 5mx, and, for a recent demonstration involving an Unmanned Aerial Vehicle, JACK was run on a Hewlett-Packard iPAQ PDA.

Application areas for JACK can be loosely categorised as:

- **Autonomous systems** (Unmanned Air Vehicles, Holonic manufacturing).
- **Modelling human-like decision making.**
- **Decision support applications** (The Collection Plan Management System (CPMS)).
- **Architectural “glue”** (Weather alerting system developed for the Australian Bureau of Meteorology).

- Many of JACK's applications are military: Usually associated with logistics (planning) and simulation, rather than with battlefield use.
- One such application is the **Collection Plan Management System (CPMS)**, which assists human in planning the deployment of surveillance and reconnaissance resources.
- The system comprises a database with:
 - Information on the terrain;
 - Information on the available resources;
 - Information on the tasks to be carried out;
 - A visualisation module;
 - A planning system written in JACK.
- The planning system presents a number of possible plans for assessment by the human experts.
- The JACK planner is structured as a collection of agents mirroring the existing command and control (C2) structure.

- Another area where JACK has been used is as architectural “glue” to connect together components of a system.
- By structuring a system as a collection of agents, one obtains a system that is more loosely coupled, and that is easier to modify and extend.
- One example of this application of JACK is the alerting system developed for the Australian Bureau of Meteorology.

- The system receives information from a range of sources including storm predictions, current observations from automated weather stations, predictions issued for the area around airports and information about bush fires.
- Various conditions, such as discrepancies between forecasts and observations, are checked for and alerts are generated.
- The system is structured as a multi-agent system where agents subscribe to information providers.
- Experiences with extending this system have been positive, for example extending the system to deal with a new type of information source only took a number of days.

- A basic property of agents is that they are autonomous, and so a natural application area for JACK is developing software that operates autonomously.
- One example is the recent use of JACK on an Unmanned Aerial Vehicle (UAV).
- The role of JACK is not to control the vehicle directly, but rather to provide higher-level decision-making about what to do next, e.g., where should the UAV fly to?
- JACK's ability to deal with failure and to flexibly achieve goals is crucial in providing the UAV with a decision making capability that allows it to be independent and robust.
- JACK's support for teams can be used to allow multiple vehicles to cooperate in achieving their goals; for example, one UAV might act as a decoy allowing another UAV with a video camera to approach undetected.

- A feasibility demonstration of JACK onboard a UAV has been done.
<http://www.agent-software.com/shared/resources/pressReleases/Avatar-JACK-F040706USb.pdf>
- Finally, **Holonic manufacturing** is another application area where JACK has been used to develop autonomous software.
- In this case, the software controls a manufacturing cell.
- The challenges in agent-based manufacturing are to support more flexible manufacturing and to be robust.

JACK: A Platform for Research

- In addition to being aimed at industrial application development, JACK has also been found to be suitable as a basis for research.
- We describe this here for two reasons:
 - Because one the goals of JACK is to “enable further applied research”;
 - Because this research has, in some cases, involved extending JACK, and so it shows that JACK can be easily extended.
- One area of research concerns making BDI agents more intelligent, or at the very least more rational.
- One issue that is shared by BDI platforms is that although a BDI agent may have multiple goals that are being pursued at a given time, no reasoning is done about the interaction between the goals.

- JACK has also been extended with look-ahead planning by integrating with JSHOP, an HTN (Hierarchical Task Network) planner written in Java.
- Work by Poutakidis *et al.* has proposed and implemented on top of JACK a debugger that automatically detects errors by monitoring messages between agents and raising an alert if the messages do not conform to the interaction protocol that is meant to be followed.

Agent-Oriented Programming Language

Functionality

- Q:** Does the language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?
- A:** The **JACK** language supports BDI style practical reasoning as well as forward-directed inference reasoning, and allows for various agent concepts such as mental attitudes, deliberation, adaptation, reactive and proactive behaviour. There is a **JACK** extension towards a Cognitive Architecture, for inclusion of cognitive parameters and variations to the reasoning processes, and for modelling of cognitive influences by behaviour moderators.

Communication

- Q: Does the language provide high-level (i.e., speech-act based) primitives for communication (as well as general addressing mechanism such as broadcast and multi-cast)?
- A: **JACK** provides high-level primitives for communication between agents. Communication is peer-to-peer, and does not include broadcast or multi-cast addressing.

Underlying Computational Model

Q: Does the language support the design of mobile agents, and if so, which kind of mobility (weak and/or strong)?

A: **JACK** is not intended for mobile agents.

Simplicity

Q: How easy it is to use and understand the language?

A: **JACK** is an easy-to-use programming language in the BDI family, and the tool suite includes graphical programming tools both for program design and for decision logic.

Preciseness

Q: Does the language have clear and precise semantics? How has it been formalised?

A: The JACK language has clear and precise (but not formal) semantics.

Expressiveness

Q: Is the language suitable for the implementation of a variety of agent-oriented programs and applications or is it purpose-specific?

A: **JACK** is a full programming language well suited for a variety of agent applications.

Extensiveness

Q: Does the language allow the definition of new language components from the basic constructs in a systematic way?

A: **JACK** allows new program elements to be defined in a systematic way, through compiler plugins.

Verification

Q: Does your approach provide a clear path for the (formal) verification of programs written in this language?

A: This has not been investigated.

Software Engineering Principles

- Q:** Have Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, etc., been considered or adopted within design of this language?
- A:** The **JACK** language is a full-flavoured programming language that combines the logic oriented BDI style with the object-oriented Java style, and it further includes programming elements providing increased support for abstraction, modularisation, information hiding and generic programming.

Language Integration

Q: Does your approach deal with the possibility of integrating the language with existing (well-known) programming language (e.g., Java)?

A: **JACK** is fully integrated with Java, and it also includes integration mechanisms for combining **JACK** agents with C++ programs.

Language Integration (continued)

Q: Can the language be interfaced with other programming languages, or does it allow the invocation of methods/programs built using other (classical) programming languages?

A: **JACK** is fully integrated with Java.

Platform

Deployment and Portability

- Q: Does the platform provide material, such as documentation, tutorials or training of any kind, installation and deployment guidelines, to help users in deploying their systems?
- A: **JACK** is well documented through a range of manuals and practicals, and is easily installed via the downloadable installer.

Deployment and Portability (continued)

Q: Does the platform require a specific computing environment (computer architecture, operating system, libraries, etc.) to be used / deployed?

A: **JACK** runs on all Java platforms from 1.1.3, and has been run on PDAs (Psion 5mx and an HP iPAQ).

Standards Compliance

- Q: To what extent does the platform adhere to the standards (FIPA, MASIF, etc.) with respects to: general architecture, naming service, white- and yellow-page services, mobility services, agent-life cycle management, etc.?
- A: The **JACK** platform is itself proprietary, but includes the standard architectural elements, and there are FIPA wrapper extensions.

Platform Extensibility

Q: Can the platform be extended with additional functionality, for example through Open Source collaboration?

A: **JACK** is built to be open, with a range of “hooks” at various levels to simplify extensions. **JACK** is not open source.

Available Tools

- Q: What tools are provided by the platform for the management, monitoring, logging and debugging of applications?
- A: **JACK** comes with several mechanisms for logging and debugging of **JACK** agent execution.

Available Tools (continued)

Q: What documentation for on-line help, and manuals for the platform's installation, use, and maintenance are available?

A: The **JACK** package includes manuals in PDF and HTML format.

Available Tools (continued)

Q: Are there tools for administration, management, and configuration of the platform? Is an IDE provided?

A: **JACK** includes a development tool.

Tool Integration

Q: In existing applications, what tools (e.g., JESS, web services, JSP) have been integrated or are known to work well with applications running on this platform ?

A: **JACK** is fully integrated with Java and all Java tools can be used.

Technical Interoperability

Q: Is an application aimed at running on this platform tied to a specific programming language, specific architectures (e.g., .NET, J2EE), or are there special operating system requirements?

A: **JACK** is not tied to any specific operation environment.

Performance Issues

- Q: What number of agents can be expected to run efficiently within a single instance of the platform, what scale of number of messages can be handled by the platform, etc.?
- A: A single process can host thousands of **JACK** agents.

Performance Issues (continued)

Q: What is the current state of the platform (simple prototype, available as a commercial product, stable Open Source distribution, etc.)?

A: **JACK** is a fully supported commercial product.

Multi-Agent Systems Features

Q: Does the platform support open multi-agent systems and heterogeneous agents?

A: **JACK** supports open multi-agent systems and heterogeneous agents.

Multi-Agent Systems Features (continued)

Q: Does the platform provide centralised or distributed control, and hierarchical structure of agents?

A: **JACK** includes a language extension for team-oriented programming, which simplifies coordinated activity and distributed control. The **JACK** Teams model includes role declarations and hierarchical, dynamic teams.

Multi-Agent Systems Features (continued)

Q: Does the platform offer libraries for programming multi-agent systems (libraries of interaction protocols, agent or group templates, reusable agent or organisation components, etc.)?

A: **JACK** does not include any pre-programmed libraries of **JACK** code.

Applications Supported by the Language and Platform

Typical Examples

Q: What types of application have already been developed with this platform (toy problems, real-world applications, industrial applications)? What are the most prominent examples?

A: **JACK** is being used for several real-world, industrial applications.

Targeted Domains

- Q:** Is any particular domain of application (e.g., simulation, resource allocation, mobile computation) targeted by your approach?
- A:** The BDI style programming is well suited to strategic robot control, as used in manufacturing plants, autonomous vehicles, and simulation, as well as business logic applications, including application of analytical procedures, compliance processes, and situated decision making.

References I



Marcus J. Huber.

JAM: A BDI-theoretic mobile agent architecture.

In Proceedings of the Third International Conference on Autonomous Agents (Agents'99), pages 236–243, 1999.



T. Juan, A. Pearce, and L. Sterling.

ROADMAP: Extending the Gaia methodology for complex open systems.

In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pages 3–10. ACM Press, 2002.



Andrew Lucas, Peter Corke, Ralph Rönquist, Pavan Sikka, Magnus Ljungberg, and Nick Howden.

Teamed UAVs – A new approach with intelligent agents.

In AIAA Unmanned Unlimited, 2003.

References II



R. Machado and R.H. Bordini.

Running AgentSpeak(L) agents on SIM_AGENT.

In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII - Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*. Springer-Verlag LNAI 2333, August 2001.



Emma Norling and Frank E. Ritter.

Embodying the JACK agent architecture.

In Markus Stumptner, Dan Corbett, and Mike Brooks, editors, *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 368–377. Springer LNAI 2256, December 2001.



Lin Padgham and Michael Winikoff.

Developing Intelligent Agent Systems: A Practical Guide.

John Wiley and Sons, 2004.

ISBN 0-470-86120-7.

References III



David Poutakidis, Lin Padgham, and Michael Winikoff.

Debugging multi-agent systems using design artifacts: The case of interaction protocols.

In Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02), pages 960–967. ACM Press, July 2002.



Talal Rahwan, Tarek Rahwan, Iyad Rahwan, and Ronald Ashri.
Agent-based support for mobile users using AgentSpeak(L).

In P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors, Agent-Oriented Information Systems (AOIS 2003): Revised Selected Papers, pages 45–60. Springer LNAI 3030, 2004.

References IV



John Thangarajah, Lin Padgham, and Michael Winikoff.

Detecting and avoiding interference between goals in intelligent agents.
In Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), 2003.



John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer.

Avoiding resource conflicts in intelligent agents.

In F. van Harmelen, editor, Proceedings of the 15th European Conference on Artificial Intelligence. IOS Press, 2002.



Steven Wilmott, Omer Rana, Karl-Heinz Krempels, Peter McBurney, and Georg Weichert.

Networked agents: Towards large-scale deployment of agents in open networked environments (NET AGENTS).

AgentLink News, 16:16–17, December 2004.

Available from <http://www.agentlink.org/newsletter/>.

References V



Franco Zambonelli and Andrea Omicini.

Challenges and research directions in agent-oriented software engineering.

Autonomous Agents and Multi-Agent Systems, 9(3):253–283, November 2004.

Abstract

Enabling effective interactions between agent teams and humans for disaster response is a critical area of research, with encouraging progress in the past few years. However, previous work suffers from two key limitations: (i) limited human situational awareness, reducing human effectiveness in directing agent teams and (ii) the agent team's rigid interaction strategies that limit team performance. This paper presents a software prototype called **DEFACTO** (Demonstrating Effective Flexible Agent Coordination of Teams through Omnipresence). **DEFACTO** is based on a software proxy architecture and 3D visualization system, which addresses the two limitations described above. First, the 3D visualization interface enables human virtual omnipresence in the environment, improving human situational awareness and ability to assist agents. Second, generalizing past work on adjustable autonomy, the agent team chooses among a variety of “team-level” interaction strategies, even excluding humans from the loop in extreme circumstances.

Keywords

Multiagent Systems, Adjustable Autonomy, Teamwork, Disaster Response.

Author

N. Schurr, J.Marecki, J.P. Lewis, M. Tambe, and P. Scerri

- We envision **future disaster response** to be performed with a **mixture** of **humans performing high level decision-making, intelligent agents coordinating the response** and **humans and robots performing key physical tasks**.
- These **heterogeneous teams of robots, agents, and people** will provide the safest and most effective means for quickly responding to a disaster, such as a terrorist attack.
- A **key aspect** of such a response will be **agent-assisted vehicles working together**.

- Specifically, **agents will assist the vehicles in planning routes, determining resources to use and even determining which fire to fight.**
- Despite advances in agent technologies, **human involvement will be crucial.**
- Allowing humans to make **critical decisions** within a team of intelligent agents or robots is prerequisite for allowing such teams to be used in domains where they can cause physical, financial or psychological harm.
- These **critical decisions include:**
 - **Decisions for moral or political reasons;**
 - **Coordination decisions.**

- Human interaction with agent teams is critical in a large number of current and future applications.
- Current efforts emphasize humans collaboration with robot teams in space explorations, humans teaming with robots and agents for disaster rescue, as well as humans collaborating with multiple software agents for training.
- This paper focuses on the challenge of improving the effectiveness of applications of human collaboration with agent teams.
- Previous work has reported encouraging progress in this arena, e.g., via proxy-based integration architectures, adjustable autonomy and agent-human dialogue.

- Despite this encouraging progress, previous work suffers from two key limitations:
 - First, when interacting with agent teams acting remotely, human effectiveness is hampered by interfaces that limit their ability to apply decision-making skills in a fast and accurate manner.
 - Techniques that provide telepresence via video are helpful, but cannot provide the global situation awareness.
 - Second, agent teams have been equipped with adjustable autonomy (AA) but not the flexibility critical in such AA.

- Indeed, the appropriate AA method varies from situation to situation.
- In some cases the human user should make most of the decisions.
- In other cases human involvement may need to be restricted.
- Such flexible AA techniques have been developed in domains where humans interact with individual agents, but whether they apply to situations where humans interact with agent teams is unknown.

- **DEFACTO** (Demonstrating Effective Flexible Agent Coordination of Teams through Omnipresence).
- Software prototype system, **DEFACTO** , that enables agent-human collaboration and addresses the issues of enhanced user interface and flexible adjustable autonomy.
- The user interface and proxy-based teamwork (called **Machinetta**) are incorporated in **DEFACTO** .

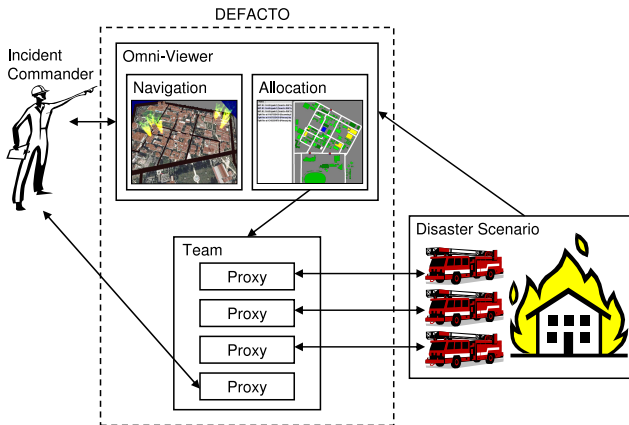


Figure 58: DEFACTO system applied to a disaster rescue.

- The Omni-Viewer is an advanced human interface for interacting with an agent-assisted response effort.
- The Omni-Viewer provides for both global and local views of an unfolding situation, allowing a human decision-maker to precisely assess the information required for a particular decision.
- A team of completely distributed proxies, where each proxy encapsulates advanced coordination reasoning based on the theory of teamwork, controls and coordinates agents in a simulated environment.

- The use of the proxy-based team brings realistic coordination complexity to the prototype and allows more realistic assessment of the interactions between humans and agent-assisted response.
- Currently, we have applied **DEFACTO** to a disaster rescue domain.
- The incident commander of the disaster acts as the **human user** of **DEFACTO** .
- This disaster can either be “man made” (terrorism) or “natural” (earthquake).

- We focus on two urban areas: a square block that is densely covered with buildings (we use one from Kobe, Japan) and the University of Southern California (USC) campus, which is more sparsely covered with buildings.
- In our scenario, several buildings are initially on fire, and these fires spread to adjacent buildings if they are not quickly contained.
- The goal is to have a human interact with the team of fire engines in order to save the most buildings.
- While designed for real world situations, **DEFACTO** can also be used as a training tool for incident commanders when hooked up to a simulated disaster scenario.

- To provide flexible AA, we generalize the notion of *strategies* from single-agent single-human context.
- In our work, agents may flexibly choose among team strategies for adjustable autonomy instead of only individual strategies; thus, depending on the situation, the agent team has the flexibility to limit human interaction, and may in extreme cases exclude humans from the loop.

- Finally, we present results from detailed experiments with **DEFACTO** in Robocup Rescue domain, which reveal two major surprises.
- First, contrary to previous results, human involvement is not always beneficial to an agent team— despite their best efforts, humans may sometimes end up hurting an agent team's performance.
- Second, increasing the number of agents in an agent-human team may also degrade the team performance, even though increasing the number of agents in a pure agent team under identical circumstances improves team performance.
- Fortunately, in both the surprising instances above, DEFACTO's flexible AA strategies alleviate such problematic situations.

- **DEFACTO** is currently instantiated as a prototype of a future disaster response system.
- **DEFACTO** has been repeatedly demonstrated to key police and fire department personnel in Los Angeles area, with very positive feedback.

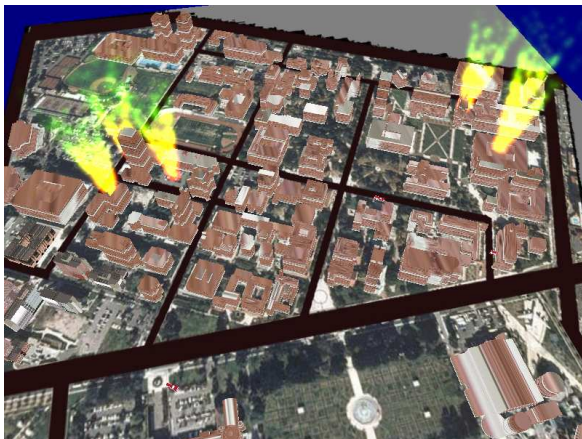


Figure 59: Omni-Viewer during a scenario: (a) Multiple fires start across the campus.



Figure 60: Omni-Viewer during a scenario: (b) The Incident Commander uses the Navigation mode to quickly grasp the situation.



Figure 61: Omni-Viewer during a scenario: (c) Navigation mode shows a closer look at one of the fires.



Figure 62: Omni-Viewer during a scenario: (d) Allocation mode is used to assign a fire engine to the fire.



Figure 63: Omni-Viewer during a scenario: (e) The fire engine has arrived at the fire.



Figure 64: Omni-Viewer during a scenario: (f) The fire has been extinguished.

- Our goal of allowing fluid human interaction with agents requires a visualization system that provides the human with a global view of agent activity as well as showing the local view of a particular agent when needed.
- We have developed an **omnipresent viewer, or Omni-Viewer**, which will allow the human user diverse interaction with remote agent teams.
- A global view is obtainable from a two-dimensional map.
- A local perspective is best obtained from a 3D viewer.

- The Omni-Viewer incorporates both a conventional map-like 2D view, Allocation Mode (Figure d) and a detailed 3D viewer, Navigation Mode (Figure a).
- The Allocation mode shows the global overview as events are progressing and provides a list of tasks that the agents have transferred to the human.
- The Navigation mode shows the same dynamic world view, but allows for more freedom to move to desired locations and views.
- The user can drop to the virtual ground level, thereby obtaining the world view (local perspective) of a particular agent.
- At this level, the user can “walk” freely around the scene, observing the local logistics involved as various entities are performing their duties.
- This can be helpful in evaluating the physical ground circumstances and altering the team’s behavior accordingly.

- We assume that a high resolution 3D model has already been created and the only data that is transferred during the disaster are important changes to the world.
- Generating this suitable 3D model environment for the Navigation mode can require months or even years of manual modeling effort, as is commonly seen in the development of commercial video-games.
- To avoid this level of effort we make use of the work of You et. al. in rapid, minimally assisted construction of polygonal models from LiDAR (Light Detection and Ranging) data.
- Given the raw LiDAR point data, we can automatically segment buildings from ground and create the high resolution model that the Navigation mode utilizes.
- The construction of the USC campus and surrounding area required only two days using this approach.
- LiDAR is an effective way for any new geographic area to be easily inserted into the Omni-Viewer.

- We use the JME game engine to perform the actual rendering due to its cross-platform capabilities.
- JME is an extensible library built on LWJGL (Light Weight Java Game Library), which interfaces with OpenGL and OpenAL.
- This environment easily provides real-time rendering of the textured campus environment on mid-range commodity PCs.
- JME utilizes a scene graph to order the rendering of geometric entities.
- It provides some important features such as OBJ format model loading and also various assorted effects such as particle systems for fires.

Proxy-based teamwork

- In disaster rescue domains robust multi agent teams are more likely to perform better than centralized approaches.
- **DEFACTO** is built on the state-of-the-art multi agent infrastructure called **Machinetta**.
- The modular structure of Machinetta's main components and the fact that it provides coordination algorithms rather than fixed multi-agent infrastructure ensures its versatility which contributes to the reusability of **DEFACTO** for different domains.
- The robustness of Machinetta is achieved through decentralized role allocation, communication and coordination algorithms which use the concept of moving agents instead of fixed messages.

- A key hypothesis in this work is that intelligent distributed agents will be a key element of a future disaster response.
- A critical role of these intelligent agents will be to manage coordination between all members of the response team.
- Specifically, we are using coordination algorithms inspired by theories of teamwork to manage the distributed response.

- The general coordination algorithms are encapsulated in **proxies** with each proxy representing one team member in the team.
- Machinetta Proxies, which extend the successful Teamcore proxies are implemented in Java and are freely available on the web.
- The concept of a reusable proxy differs from many other “multiagent toolkits” in that it provides the coordination **algorithms**, e.g., algorithms for allocating tasks, as opposed to the **infrastructure**, e.g., APIs for reliable communication.

- Currently, **DEFACTO** is applied to a **Robocup Rescue domain** which incorporates detailed disaster simulator as well as templates for three types of agents:
 - Fire Engines;
 - Ambulances;
 - Police Cars.
- At this stage of the system development we focus on Fire Engines and simulate only the fire spread and building damage.
- Agents in our simulation are Fire Engines taking on new Fight Fire requests and reporting the status of buildings.

Main aspects of these agents are:

1. Pro-activeness:

- Each agent stores a list of plans it is able to perform and whenever plan preconditions are met, roles associated with plans are immediately triggered;
- Agents pro-activeness can be varied through adjustable autonomy strategies resulting in the increased performance of the whole team.

2. Reactivness:

- Each agent moves around the environment, scans it for emerging fires and reports the status of the buildings on fire;
- In case of an environment change agent's first task is to communicate the news to other team members and consequently establish the basis for a new **Fight Fire** plan.

Main aspects of these agents are:

3. Mobility:

- Agent movement affects its sensing of the environment and choice of which fire to fight first;
- Priority is given to the closest burning building;
- Agents are susceptible to road congestion generated in real time by the traffic simulator.

4. Configurability:

- Agents can have flexible level of intelligence depending on the contents of their declarative configuration files which store agent beliefs, plans, adjustable autonomy strategies etc.

5. Flexible architecture:

- The modular structure of Machinetta Proxies allows them to be reused for different domains with interchangeable coordination algorithms.

Adjustable Autonomy

- In this paper, we focus on a key aspect of the proxy-based coordination: **Adjustable Autonomy**.
- Adjustable autonomy refers to an agent's ability to dynamically change its own autonomy, possibly to transfer control over a decision to a human.
- Previous work on adjustable autonomy could be categorized as either involving a single person interacting with a single agent or a single person directly interacting with a team.
- In the single-agent single-human category, the concept of flexible transfer-of-control strategy has shown promise.

- A transfer-of-control strategy is a preplanned sequence of actions to transfer control over a decision among multiple entities.

Example 8.1

For example, an AH_1H_2 strategy implies that an agent (A) attempts a decision and if the agent fails in the decision then the control over the decision is passed to a human H_1 , and then if H_1 cannot reach a decision, then the control is passed to H_2 .

- Since previous work focused on single-agent single-human interaction, strategies were individual agent strategies where only a single agent acted at a time.

- An optimal transfer-of-control strategy optimally balances the risks of not getting a high quality decision against the risk of costs incurred due to a delay in getting that decision.
- Flexibility in such strategies implies that an agent dynamically chooses the one that is optimal, based on the situation, among multiple such strategies ($H_1 A$, AH_1 , $AH_1 A$, etc.) rather than always rigidly choosing one strategy.
- The notion of flexible strategies has not been applied in the context of humans interacting with agent-teams.
- A key question is whether such flexible transfer of control strategies are relevant in agent-teams, particularly in a large-scale application such as ours.

- **DEFACTO** aims to answer this question by implementing transfer-of-control strategies in the context of agent teams.
- The strategies are not limited to individual agent strategies, but also enables team-level strategies

Example 8.2

For example, rather than transferring control from a human to a single agent, a team-level strategy could transfer control from a human to an agent-team.

- Each proxy is provided with all strategy options; the key is to select the right strategy given the situation.

Example 8.3

- An example of a team level strategy would combine A_T Strategy and H Strategy in order to make A_TH Strategy.
- The default team strategy, A_T , keeps control over a decision with the agent team for the entire duration of the decision.
- The H strategy always immediately transfers control to the human.
- A_TH strategy is the conjunction of team level A_T strategy with H strategy.
- This strategy aims to significantly reduce the burden on the user by allowing the decision to first pass through all agents before finally going to the user, if the agent team fails to reach a decision.

- The Machinetta software consists of five main modules, three are domain independent and two are tailored for specific domains.
- The three domain independent modules are for coordination reasoning, maintaining local beliefs (state) and adjustable autonomy.
- The domain specific modules are for communication between proxies and communication between a proxy and a team member.
- The modules interact with each other only via the local state with a blackboard design and are designed to be “plug and play”.

- The coordination reasoning is responsible for reasoning about interactions with other proxies.
- The adjustable autonomy algorithms reason about the interaction with the team member, providing the possibility for the team member to make any coordination decision instead of the proxy.
- In practice, the overwhelming majority of coordination decisions are made by the proxy, with only key decisions referred to team members.

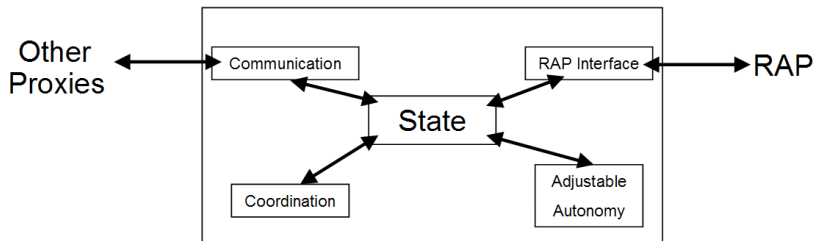


Figure 65: Proxy Architecture

Machinetta Proxy components

- **Communication:** Communication with other proxies;
- **Coordination:** Reasoning about team plans and communication;
- **State:** The working memory of the proxy;
- **Adjustable Autonomy:** Reasoning about whether to act autonomously or pass control to the team member;
- **RAP Interface:** Communication with the team member.

- Teams of proxies implement **team oriented plans** (TOPs) which describe joint activities to be performed in terms of the individual **roles** to be performed and any constraints between those roles.
- Typically, TOPs are instantiated dynamically from TOP templates at runtime when preconditions associated with the templates are filled.
- Typically, a large team will be simultaneously executing many TOPs.

Example 8.4

- For example, a disaster response team might be executing multiple fight fire TOPs.
- Such fight fire TOPs might specify a breakdown of fighting a fire into activities such as checking for civilians, ensuring power and gas is turned off and spraying water.

- Constraints between these roles will specify interactions such as required execution ordering and whether one role can be performed if another is not currently being performed.
- TOPs do not specify the coordination or communication required to execute a plan, the proxy determines the coordination that should be performed.

Organisation

- The **proxy based approach to coordination** is **completely distributed**, with each proxy working closely with the team member it represents in the environment.
- Coordination requires that **joint intentions** and **mutual beliefs** are **formed in a distributed manner** before the team begins executing a task.
- Agents are only required to form **joint intentions** and **mutual beliefs** with other team members with whom they are directly collaborating on a sub-goal, rather than with the whole team.
- This leads to the possibility of the team performing duplicate or conflicting tasks.
- Hence additional conflict resolution algorithms are required to remove these conflicts.

- Underlying the conflict resolution mechanism is a static, logical network involving all the members of the team.
- This network is referred to as the **associates network**.
- Team members share critical information with their neighbors in the associates network.
- By ensuring that the associates network has a **small worlds** property, there is very high probability that at least one agent will get to know of any conflict and can initiate a resolution process.

- Team members collaborating on a sub-plan form a sort of sub-team with their neighbors in the associates network by virtue of their joint intention and mutual beliefs.
- The members of a sub-team, and indeed the sub-plans, can change over time, resulting in an emergent organizational structure consisting of dynamically changing, overlapping sub-teams.
- No hierarchy or centralized control of any type is present in Machinetta.

- The algorithms in Machinetta are designed to be very scalable, allowing large teams to be deployed to achieve complex tasks.
- There is no need for yellow pages type services, because team members need only interact with neighbors in the network.
- Provided new team members can integrate themselves successfully into the network, maintaining the network's small worlds property, Machinetta can support dynamic addition of new team members.

Interaction

- When developing large teams, protocol and software robustness is critically important.
- When hundreds of distributed team members asynchronously coordinate to simultaneously achieve hundreds of sub-plans over a period of time, any “bugs” in the interaction code will be inevitable found.
- We developed a novel way of implementing interaction that is particularly robust and relatively easy to debug.
- We are exploring the use of mobile agents that transfer both coordination state and a message as they move between proxies.

- The use of mobile agents for coordination leads to high degrees of robustness in at least two key ways:
 - ① It is easier to develop reliable means to know whether messages are “lost”, since the agent itself can ensure its own movement around the team;
 - ② Coordination algorithms are simpler to implement because they are entirely encapsulated within the code of a single mobile agent.
- Management of interaction state and handling of coordination failures, etc., is greatly simplified.

- The use of mobile agents means that the proxies can be thought of as a type of mobile agent platform.
- The proxies are active in providing information to the mobile agents and even, when the adjustable autonomy decides a human should make a decision, making decisions on the mobile agents behalf.
- It is possible to think of the coordination as being implemented by mobile agents moving around a small worlds network of active mobile agent platforms.

MAS Environment

- Machinetta is designed to be both domain independent and to work with highly heterogeneous teams.
- It has been possible to demonstrate Machinetta in several simulation environments.
- Machinetta has been used for coordination of high fidelity simulations of search and rescue robots.
- The most stringent tests of Machinetta's coordination capabilities will come in late 2005 when it is used for an Air Force flight test involving three simulated unmanned aerial vehicles (UAVs) and one real UAV.
- Machinetta can effectively coordinate large numbers of UAVs to efficiently execute a wide area search and destroy mission with sufficiently low communication bandwidth to be feasible in a military domain.

- The domains in which Machinetta have been used, though varied, share some common traits.
- The domains have typically allowed complex tasks to be broken down into smaller subtasks with most of the required coordination being for specific subtasks rather than across subtasks.
- Machinetta has been required to deal with dynamics in all the domains in which it have been used but explicit adversarial reasoning has not been performed.

- We have conducted experiments with the **DEFACTO** system connected to the Robocup Rescue simulation environment.
- All the simulation components were running on one desktop with two AMD 1.8 GHz processors and 4GB of ram.
- The following processes were simulated on the desktop:
 - All the Machinetta proxies and their communication server;
 - Robocup rescue kernel including traffic, blockade, fire and building collapse simulators;
 - Allocation viewer.

Evaluation

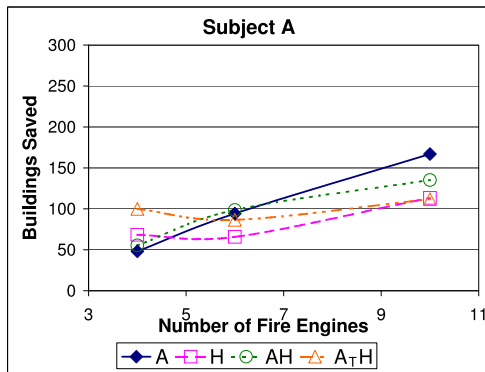


Figure 66: Performance of subjects.

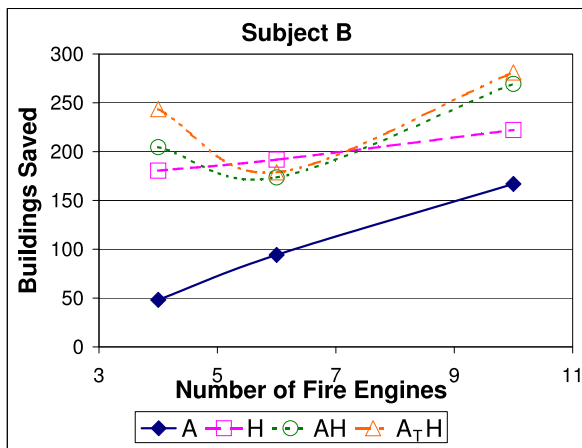


Figure 67: Performance of subjects.

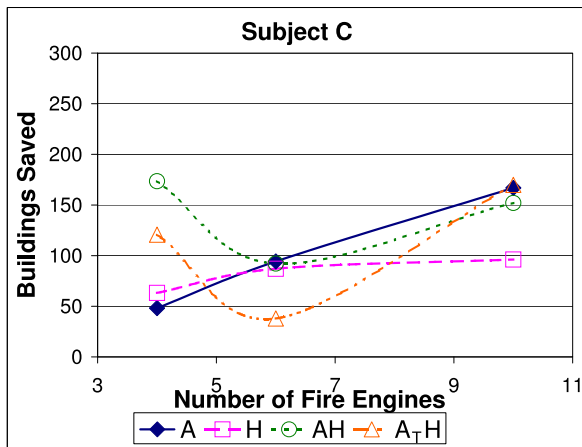


Figure 68: Performance of subjects.

- **Human involvement** with agent teams **does not necessarily lead to improvement** in **team performance**.
- **Providing more agents** at a human's command **does not necessarily improve** the agent team performance
- **No strategy dominates** through all the experiments given varying numbers of agents.
- **Complex team-level strategies are helpful** in practice.

Strategy	H			AH			A_TH		
# of agents	4	6	10	4	6	10	4	6	10
Subject 1	91	92	154	118	128	132	104	83	64
Subject 2	138	129	180	146	144	72	109	120	38
Subject 3	117	132	152	133	136	97	116	58	57

Table 1: Total amount of allocations given.

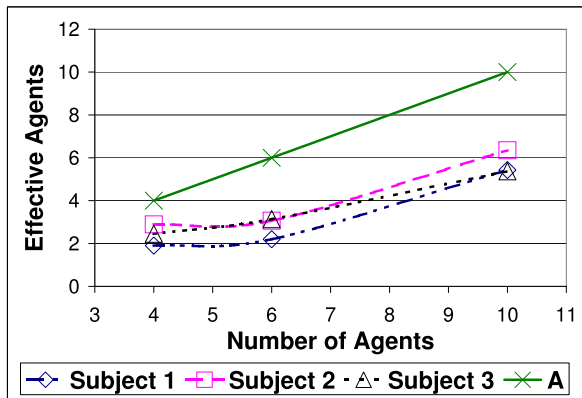


Figure 69: AG_H performance

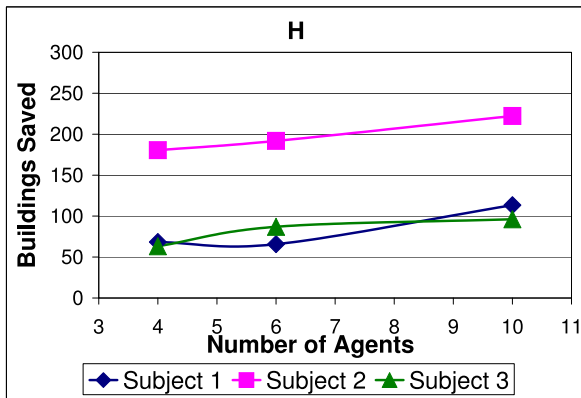


Figure 70: *H* performance

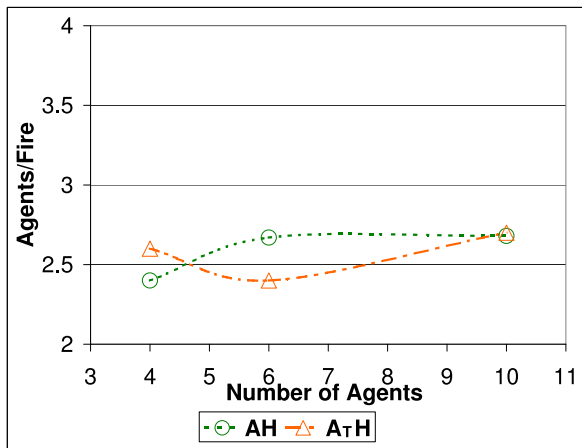


Figure 71: Amount of agents per fire assigned by subjects 1.

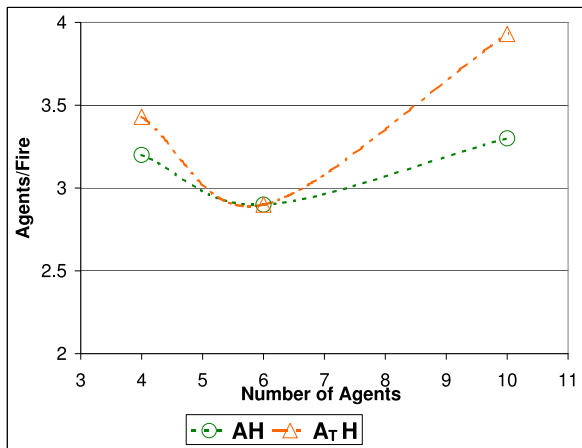


Figure 72: Amount of agents per fire assigned by subjects 2.

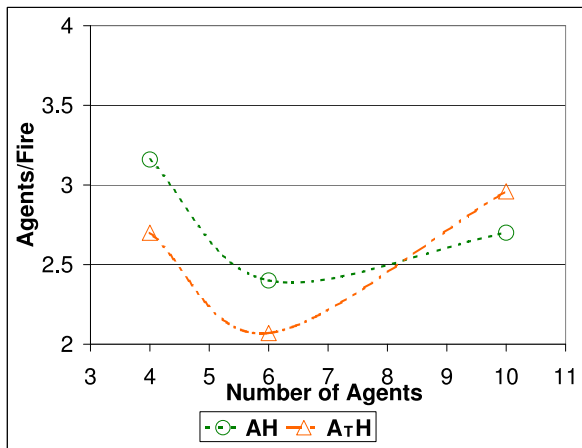


Figure 73: Amount of agents per fire assigned by subjects 3.

References I



James F. Allen.

The TRAINS project: A case study in building a conversational planning agent.

Journal of Experimental and Theoretical AI (JETAI), 7:7–48, 1995.



Mark H. Burstein, Alice M. Mulvehill, and Stephen Deutsch.

An approach to mixed-initiative management of heterogeneous software agent teams.

In *HICSS*, page 8055. IEEE Computer Society, 1999.



Jacob W. Crandall, Curtis W. Nielsen, and Michael A. Goodrich.

Towards predicting robot team performance.

In *SMC*, 2003.



R. Hill, J. Gratch, S. Marsella, J. Rickel, W. Swartout, and D. Traum.

Virtual humans in the mission rehearsal exercise system.

In *KI Embodied Conversational Agents*, 2003.

References II



Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsushi Shinjoh, and Susumu Shimada.

Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research.

In *IEEE SMC*, volume VI, pages 739–743, Tokyo, October 1999.



E. Liao, P. Scerri, and K. Sycara.

A framework for very large teams.

In *AAMAS'04 Workshop on Coalitions and Teams*, 2004.



D. V. Pynadath and M. Tambe.

Automated teamwork among heterogeneous software agents and humans.

Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS), 7:71–100, 2003.

References III



R. Ruddle, S. Payne, and D. Jones.

Navigating buildings in desktop virtual environments: Experimental investigations using extended navigational experience.

J. Experimental Psychology - Applied, 3(2):143–159, 1997.



P. Scerri, E. Liao, Yang. Xu, M. Lewis, G. Lai, and K. Sycara.

Theory and Algorithms for Cooperative Systems, chapter Coordinating very large groups of wide area search munitions.

World Scientific Publishing, 2004.



Ulrich Neumann Suya You, Jinhui Hu and Pamela Fox.

Urban site modeling from lidar.

In *Proc. 2nd Int'l Workshop Computer Graphics and Geometric Modeling (CGGM)*, pages 579–588, 2003.

Abstract

ARTIMIS is an effective intelligent agent technology designed and developed by France Telecom. It provides a generic framework to instantiate dialogue agents that are able to engage in rich interactions with human users (with no restriction on the communication media) as well as with other software agents. Several operational **ARTIMIS**based applications have been developed, and commercial services have begun to be deployed.

ARTIMIS relies on the principle that a system's ability to carry on a natural dialogue with a human user must result from the system's inherent intelligence. Consequently, an intelligent dialogue system needs to be first conceived and designed as an intelligent system.

ARTIMIS provides a generic framework for the development of intelligent agents whose behaviour is wholly driven by explicit cognitive principles, such as rationality, communication and cooperation. **ARTIMIS** agents can also be components of multi-agent systems. In this case, to interact with other agents, they use the FIPA ACL standard (Agent Communication Language), whose formalism and semantics come from ARCOL, the **ARTIMIS** Communication Language.

Keywords

Rational agent technology, natural human-computer dialogue, multi-agent context, inter-agent communication language, effective applications.

Author

David Sadek

- ARTIMIS is a **rational agent technology** designed and developed by France Telecom.
- It provides a **generic framework** to instantiate **dialogue agents**.
- They are able to engage in rich **interactions** with **human users** as well as with other **software agents**.
- The foundational work underlying ARTIMIS technology handles communication as a **special case of intelligent behaviour**:
An **intelligent dialogue system** has to be first an **intelligent system**.

- The necessity of **machine intelligence** obviously appears in contexts that require, at the same time, a **complex and user-friendly interaction** with a human being.
- This is overtly illustrated in natural **human-computer dialogue**.
- It is not anodyne that the **Turing test requires a dialogue situation**.
- The capabilities underlying dialogue phenomenon (perception, reasoning, learning, etc.) are required by other so-called **intelligent behaviours**, namely behaviours that display problem solving capacities.

- The ability to engage into a dialogue relies on a common ground that endows it with a non-primitive character.
- **Rational behaviour** appears as the most consensual manifestation of intelligence.
- To behave rationally is to be driven by principles that select in an optimal way the actions leading to futures compliant with a given set of motivations and goals.

- The approach underlying **ARTIMIS** technology right away targets **real systems**.
- It has therefore the ambition to cover at the theoretical level the different components of rational, communicative, and social behaviours of an agent.
- In an original way, it unifies under the same point of view the problem of designing dialogue systems and that of designing intelligent artificial agents.
- This approach points out the paradigm of rational dialogue agent as the one that has to ground the construction of intelligent systems.
- It is intended to provide the robustness required from an intelligent system, namely to adequately react to complex situations or incompletely specified ones when the system is designed.

- **ARTIMIS** intelligent agent technology de facto strongly relates to end-user applications.
- When instantiated in a human-agent interaction context, **ARTIMIS** allows for implementing interactive services that enable natural dialogue with human users, whatever the communication media is.
- The resulting dialogue agents display advanced functionalities:
 - Natural language;
 - Mixed-initiative interaction;
 - Request and response negotiation;
 - Cooperative reactions;
 - etc.

- **ARTIMIS** can also be used in multi-agent environments, namely contexts involving several interacting software agents.
- Along these lines, its use field is extended to mediation applications, in which an **ARTIMIS** agent can manage, in a unified and cooperative way, the relationship between a user and regular information sources.
- The main offered functionalities are:
 - Identification of proposals that best fit the user's request;
 - Construction of proposals possibly combining information coming from different sources;
 - Suggestion of alternative solutions;
 - etc.



Figure 74: **ARTIMIS**: a wide range of applications

- **ARTIMIS** range of potential applications is very wide.
- It encompasses **various application domains**:
 - Personal or public assistants;
 - Information and transaction services;
 - Telecommunication services (hotlines, portals,...);
 - Directories (yellow pages, tourist guides,...);
 - Banking and financial services (stocks, account management,...);
 - E-commerce;
 - E-learning and education;
 - Intelligent interfaces for Internet search engines;
 - Games (intelligent communicating clones and avatars);
 - etc.

Formal rational interaction theory

The **ARTIMIS** approach of rational interaction is governed by two main ideas:

- 1 For an agent, engaging and following up a dialogue with a human interlocutor or with another agent can be totally justified by rational behaviour principles.
It is not necessary for an agent to have a structural dialogue model or a communication protocol, since the instances of such a model or a protocol are dynamically retrieved by rational behaviour.
- 2 A single formal theory can account for the different aspects of an agent's cognitive and communicative behaviour.

- The **ARTIMIS model** is a **formal theory of rational interaction** expressed in a **homogeneous logical framework**.
- The **basic concepts** of this framework are **mental attitudes and actions**.
- The theory shows how advanced behaviours result from the application of **primitive principles** and from their **combination**.
- It is expressed within a **unique formal model (a quantified modal logic, with a well-defined semantics)** and **modularly** constructed in different components.

Basic formal concepts

- The formal account is expressed in a first order modal language with identity.
- For the sake of brevity, only the aspects of the formalism used in this chapter are introduced in the following.

Definition 9.1

- Symbols $\neg, \wedge, \vee, \Rightarrow$, and \Leftrightarrow represent classical connectives of negation, conjunction, disjunction, logical implication and equivalence.
- \forall and \exists are respectively the universal and existential quantifiers.
- x, y , and z are symbols for individual variables.
- a, a_1 , and a_2 are variable schemata denoting action expressions.
- p and p' are taken to be closed formulae (denoting propositions), ϕ, ψ and δ formula schemata, and i, j and h variable schemata denoting agents.
- Notation $\models \phi$ means that formula ϕ is valid.

- The **concept of action (or event)** and the **basic mental attitudes** of **belief**, **uncertainty** and **choice** (or, to some extent, goal) are modelled.

Definition 9.2

The mental attitudes are respectively formalised by the modal operators:

Belief: B ; **Uncertainty:** U ; **Choice:** C

Example 9.3

Formulae as $B_i p$, $U_i p$ and $C_i p$ can be respectively read:

- “ i (implicitly) believes (that) p ”;
- “ i thinks that p is more likely than $\neg p$ ”;
- “ i desires that p currently holds”.

- The **logical rules** that express their **inherent properties and relationships** are formalised.
- The **resulting agents** are **fully introspective** and have **consistent beliefs**.
- An **agent's uncertainties** are **not closed** under logical consequence.

Definition 9.4 (Intention)

- **Intention** is defined as a **complex combination of belief and choice**.
- It is formalised by the modal operator I .
- Formula as $I_i p$ can be read “ i brings about p ”.
- The definition of intention entails that it necessarily generates a **planning process**.
- It also imposes that an agent cannot bring about a situation if the agent believes that the situation already holds:

$$\models I_i \phi \Rightarrow B_i \neg \phi$$

Definition 9.5 (Complex action expressions)

As far as *action* is concerned, complex action expressions are **recursively built** over the set of *primitive actions* (or events) using:

- A *sequence operator* $(a_1; a_2)$;
- A *nondeterministic choice operator* $(a_1 \mid a_2)$;
- An *iteration operator* (a^*) .

Definition 9.6 (Sequence)

A sequence can be formed with a single agent that may be the *void* event.

Definition 9.7 (Introspection action)

- The **introspection action**, noted $\langle i, \phi? \rangle$, is a particular event that can be performed **if and only if** agent i explicitly believes ϕ .
- It enables the agent to handle **conditional steps within plans**.

Definition 9.8 (Feasible, Done)

In order to reason about action, two modal operators are introduced, a being an action expression and ϕ a formula:

- $Feasible(a, \phi)$ means that a can take place and if it does, ϕ will be true after that;
- $Done(a, \phi)$ means that a has just taken place and ϕ was true before that.

$Agent(i, a)$ means that i denotes the only agent of the events (or actions) appearing in a .

Definition 9.9

The following abbreviations are used, *True* being the propositional constant always true:

$Feasible(a)$	\equiv	$Feasible(a, True)$
$Done(a)$	\equiv	$Done(a, True)$
$Bif_i\phi$	\equiv	$B_i\phi \vee B_i\neg\phi$
$Bref_i\delta(x)$	\equiv	$(\exists y)B_i(\text{!}x\delta(x) = y)$
$Uif_i\phi$	\equiv	$U_i\phi \vee U_i\neg\phi$
$Uref_i\delta(x)$	\equiv	$(\exists y)U_i(\text{!}x\delta(x) = y)$
$AB_{n,i,j}\phi$	\equiv	$B_iB_jB_i\dots\phi$

Definition 9.10

In the forth and sixth abbreviations, ι is the operator for definite description, defined as a term producer as follows:

$$\phi(\iota x \delta(x)) \equiv \exists y \phi(y) \wedge \delta(y) \wedge \forall z (\delta(z) \Rightarrow z = y)$$

- $\iota x \delta(x)$ is read “the (x which is) δ ”.
- $Bref_i \delta(x)$ means that agent i (thinks that she/he/it) knows the (x which is) δ .
- $Uif_i \phi$ means that either agent i is uncertain about ϕ or that he is uncertain about $\neg \phi$.
- $Uref_i \delta(x)$ has the same meaning as $Bref_i \delta(x)$, except that agent i has an uncertainty attitude with respect to $\delta(x)$ instead of a belief attitude.
- In the last abbreviation, which introduces the concept of **alternate beliefs**, n is a positive integer representing the number of B operators alternating between i and j .

There is **no restriction** on the possibility of **embedding mental attitude or action operators**.

Example 9.11

Formula $U_i B_j I_j Done(a, B_i p)$ informally means that agent i believes that, probably, agent j thinks that i has the intention that action a be done before which i has to believe p .

Definition 9.12

A **fundamental property** of the proposed logic is that the **modelled agents** are **perfectly in agreement with** their **own mental attitudes**. Formally, the following property holds:

$$\models \phi(i) \Leftrightarrow B_i \phi$$

where $\phi(i)$ is a formula governed by a modal operator formalising a mental attitude of agent i .

Rationality principles

- The second component of the theory deals with the **properties that relate an agent's intentions** with the actions enabling it to achieve them.
- It formally establishes the axioms that ground an **agent's rational behaviour**, namely the **rationality principles**.
- These principles enable an agent to generate **intentional action**.

- The components of an **action model**, in particular, of a **communicative act (CA) model** that are involved in a planning process characterise:
 - The reason for which the action is selected;
 - The conditions that have to be satisfied for the action to be planned.
- For a given action, the former is referred to as the **rational effect (RE)**, and the latter as the **feasibility preconditions (FP)**, or the **qualifications** of the action.

- Two rationality principles relate an agent's intention to its plans and actions.
- The planning process is driven by their alternate use.

Definition 9.13 (Rationality principles)

Formally, the two rationality principles are stated as follows:

$$1. \models I_i p \Rightarrow I_i \text{Done}(a_1 \mid \dots \mid a_n)$$

where a_k (k ranging from 1 to n) are all the actions such that:

① agent i knows action a_k ;

② p is the rational effect of a_k (i.e., the reason for which a_k is planned);

③ $\neg I_i \neg \text{Done}(a_k)$

$$2. \models I_i \text{Done}(a) \Rightarrow B_i \text{Feasible}(a) \vee I_i B_i \text{Feasible}(a)$$

1. The first principle gives an agent the **capability of planning an act** whenever the agent intends to achieve its **RE**.
It states that an agent's intention to achieve a given goal generates its intention that be done one of the acts:
 - ❶ Known to the agent;
 - ❷ Whose **RE** corresponds to the agent's goal;
 - ❸ That the agent has no reason for not doing them.

2. The second principle imposes on an agent, whenever it selects an action, to seek the satisfiability of its **F**Ps.

It states that an agent having the intention that some action be done, adopts the intention that the action be feasible, unless it believes that it is already feasible.

Note: The rationality principles specify a planning algorithm — **with no need for any external plan calculus mechanisms** — that deductively generates action plans, through the inference of causal chains of intentions.

Communicative act models

- The third component of the theory underlying ARTIMIS gives a **fine-grained account of a set of primitive communicative acts**, by precisely determining the **preconditions and effects** that characterise each one of them and endow it with its semantics.
- **Communicative acts** are viewed as ordinary actions, handled by the rationality principles in a regular way.

- A model of rational action should specify the feasibility preconditions and the rationale of the action.
- The expression of such a model is, in general, complex for two main reasons:
 - 1 The set of action qualifications is potentially infinite.
 - 2 The effect of an action on the world is strongly context-dependent and cannot be formulated in general terms; furthermore, “summarising” what an action should leave unchanged is a difficult problem.

- A solution that goes round the problem of effect specification is directly related to the expression of the rationality principles.
- If it is not possible to specify the actual effects of an action, it is yet possible to state what is expected from an action, that is, what are the reasons for which the action has been selected.
- This is exactly what is expressed by the first rationality principle.
- This semantics for action effect, within the framework of a model of rational behaviour, allows one to overcome the problem of effect unpredictability.

- The set of **feasibility preconditions** for a **CA** can be split into two subsets:
 - 1 The **ability preconditions**;
 - 2 The **context-relevance preconditions**.

- The **ability preconditions** characterise the intrinsic ability of an **agent** to perform a given CA.

Example 9.14

To **sincerely assert** some proposition p , an agent has to believe that p .

- The **context-relevance preconditions** characterise the relevance of the act with respect to the context in which it is to be performed.

Example 9.15

An agent can be intrinsically able to make a promise while believing that the promised action is not needed by the addressee.

Definition 9.16

The specification of an action's feasibility preconditions and rational effect is axiomatised within the logical theory through the two following properties:

$$\begin{aligned} B_i(\textit{Feasible}(a) &\Leftrightarrow \textit{FP}(a)) \\ B_i(\textit{Done}(a) &\Rightarrow \textit{RE}(a)) \end{aligned}$$

Example 9.17

A simplified model of the communicative act of **informing** about the truth of a proposition:

$$\langle i, \text{Inform}(j, \phi) \rangle$$

$$FP : B_i \phi \wedge \neg B_i B_j \phi$$

$$RE : B_j \phi$$

This model is directly axiomatised within the logical theory through the rationality principles and the following schema:

$$B_h(\text{Feasible}(\langle i, \text{Inform}(j, \phi) \rangle)) \Leftrightarrow B_i \phi \wedge \neg B_i B_j \phi$$

$$B_h(\text{Done}(\langle i, \text{Inform}(j, \phi) \rangle)) \Rightarrow B_j \phi$$

Note: Actions are not handled as data structures by a planning process, but have a logical semantics within the theory itself.

Belief reconstruction

- The fourth component of the theory deals with the **evolution of agent's beliefs**, in particular as a consequence of observing communicative actions.
- It formalises the **belief reconstruction** process and mainly deals with the converse process of **CA** planning, namely the **CA consummation** process.
- An agent comes to the conclusion that an action (or an event) it has just observed has really occurred, only under certain **admission** conditions.

Definition 9.18

- An agent considers that an action a realised by what it has just observed has occurred, only if what the agent believed before the occurrence of a is consistent with the fact that a was feasible;
- Otherwise the agent rejects a or puts into question the admissibility of the previous event(s) that make(s) a unfeasible.

Definition 9.19

- After an observation,
- **whenever an agent admits an action (or an event) corresponding to the observation, and only in this case,**
- it is necessarily committed to believe that its effects and persistent qualifications hold.

Definition 9.20

- As far as **CA**s are concerned, the agent has to come to believe that the act's performer has the intention (to make public her/his/its intention) to achieve the rational effect of the act. This is captured by the following property:

$$B_i(Done(a) \wedge Agent(j, a) \Rightarrow I_j B_i I_j RE(a))$$

- The persistent **FP**s of a **CA** are those that do not refer to time (**PFP**s).
- The following property holds:

$$B_i(Done(a) \Rightarrow PFP(a))$$

Social context

- The last component of the theory is relative to the properties enabling an agent to evolve in a **social environment**, typically a **multi-agent context**.
- It determines and formalises the **basic principles of belief and intention transfer**, and those of **cooperative behaviour**, together with a **set of cognitive properties** endowing an agent with the motivations to harmoniously react to its environment solicitations.
- A **minimal cooperation is required** for communication to be possible.

Example 9.21

- Suppose that an agent i asks an agent j if proposition p is true.
- If both agents respect the semantics of the communication language, j knows that i intends to know if p is true.
- But, without a minimal cooperation, j is in no way constrained to react to i 's request.

Minimal principle of cooperation (informal)

The **minimal principle of cooperation** states that agents must:

- React when they are addressed;
- Adopt the interlocutor's intention whenever they recognise it, and if they have no objection to adopt it.

In other words:

- **if an agent i believes that an agent j intends to achieve property p , and that itself does not have an opposite intention, then i will adopt the intention that j will (eventually) come to believe p .**

Definition 9.22 (Minimal principle of cooperation)

The **Minimal principle of cooperation** formalised by the validity of the following property:

$$B_i I_j \phi \wedge \neg I_i \neg \phi \Rightarrow I_i B_j \phi$$

- **In particular:** If an agent i thinks that an agent j is expecting something from it (and i has no objections for doing it), then i adopts the intention that j will come to believe that i has done what was expected.
- Thus, from j 's point of view, agent i is cooperating.

- The minimal principle of cooperation has a far range in application.
- It may lead to cooperative behaviours that are much more complex than merely answering questions.

Example 9.23

An agent forward a request to a competent agent if it cannot answer the request by itself.

- The previous property does not ensure that agent i really believes what it will make j believe.
- **Sincerity** is an **integral part of cooperation commitments**, yet still optional in the specification of an agent in general.

Definition 9.24 (Sincerity)

In terms of **mental attitudes**, **sincerity** can be expressed as follows:

- **An agent i cannot have the intention that an agent j comes to believe that a proposition p is true without itself believing p or without having the intention to come to believe p .**
- This property translates into the validity of the following schema:

$$I_i B_j \phi \Rightarrow B_i \phi \vee I_i B_i \phi$$

- This property taken together with the previous one ensures that an agent will act sincerely, and therefore will cooperate.
- Whenever an agent i is aware of the objectives of an agent j , then, as far as possible, i will help j to achieve them.

- A **corrective answer** is produced with the intention of correcting a belief that is considered wrong.
- Such a belief is usually a presupposition inferred from the recognised communicative act.
- A **corrective intention** arises in an agent when its belief of a proposition, about which it is competent is in contradiction with that of its interlocutor.

Definition 9.25 (Corrective intention)

Formally, this property is expressed by the validity of the following schema:

$$B_i(\phi \wedge B_j\neg\phi \wedge \textit{Comp}(i, \phi)) \Rightarrow I_i B_j\phi$$

where competence is defined as follows:

$$\textit{Comp}(i, \phi) \equiv (B_i\phi \Rightarrow \phi) \wedge (B_i\neg\phi \Rightarrow \neg\phi)$$

Global functional architecture

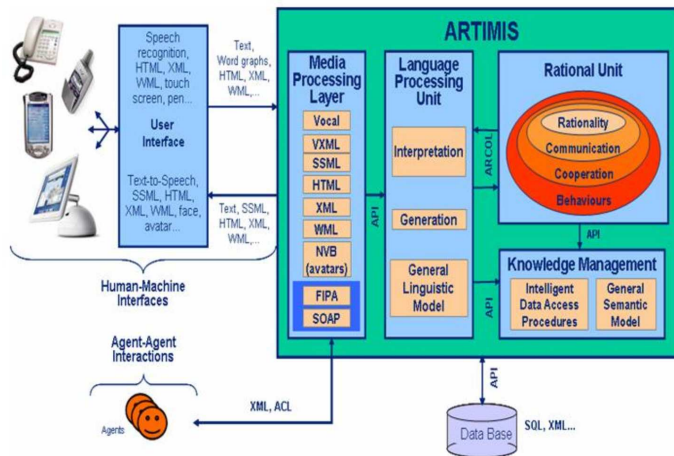


Figure 75: ARTIMIS functional architecture

Rational unit

- The **rational unit** is the **decision kernel** of the agent.
- It endows the agent with the **capability to reason about knowledge and action**.
- It performs **cooperative rational reaction calculus** producing **motivated plans of actions**, such as plans of (or including) communicative acts.
- The **communication “protocols”** are **dynamic and flexible**: there are no pre-defined inter-agent dialogue patterns.

- By itself the **rational unit is an intelligent communicating agent**.
- It can be used as a **regular communicating agent** in a **multi-agent infrastructure**.
- In the context of human-agent interaction, the **user is viewed as a particular agent**; no assumption is made about the interlocutor's type.
- The **rational unit requires natural language processing components** that bridge the gap between the human language and the internal semantic knowledge representation in terms of communicative acts with semantic contents expressed in a powerful language (a first-order modal language): **ARCOL**.

- From the **dialogue management** point of view, the rational unit can be viewed as a “**dialogue manager**”.
- Using its **generic reasoning capabilities**, it calculates the **system's reaction** to the **user's requests and responses**.
- To produce its reactions, it may also require the **Knowledge Management** component.

- The **rational unit** faithfully implements the kernel of the rational interaction theory.
- It is operated by an **inference engine** based on a “**syntactic**” approach of **deductive reasoning in first order modal logic**.
- The **axiom schemata**, of a very general scope, already pre-defined by the interaction theory, are part of an **ARTIMIS** agent's rational unit.
- The “**programmer**” can define specialised schemata for a given application.
- The set of schemata drives the rational unit reasoning process and, therefore, its reaction to its **environment solicitations**.

- **Environment solicitations** such as requests from the user or from other software agents are conveyed to the rational unit as logical sentences of the rational interaction theory.
- The **inference engine calculates the consequences** of these sentences.
- In particular, it responses or requests for precision to send to the **interlocutor** (a human user or a software agent), as well as non-communicative actions.

- For a given formula, the **inference engine** looks up if there is any **behaviour principle** that applies to this formula in order to **deduce logical consequences**.
- This procedure is then applied to the new derived consequences, until ending all the possibilities: such a procedure is a so-called **saturation-based inference algorithm**.
- Among all the consequences, the **inference engine** selects those referring to the actions (including communicative acts) to be performed by the agent, which form the agent's reactions.

Example 9.26

- Assume that after utterance analysis/interpretation the recognised communicative act is that the user or another agent wants to inform **ARTIMIS** that it wants to know if p.
- On the basis of the rationality principles, the system infers the intention of the user to know if p.
- The cooperation principles allow the system to adopt the intention that the user eventually comes to know if p.
- The system adopts the intention of informing the user that p or informing it that not p.
- The system then selects the one of these two actions that is currently feasible and transmits it to the natural language generator.
- In the case of a negative answer, **ARTIMIS** will produce, as far as possible, a suggestive response.

Knowledge management unit

This component supplies two main functionalities:

- Constraint restriction and relaxation calculus mechanisms.
- Standard interfaces (ODBC, Oracle, XML, etc.) to **external information systems**.

- The **reasoning process** carried out by the rational unit partially relies on specific application data.

Example 9.27

If an **ARTIMIS** Agent is intended to provide a train schedule, it needs to have data about the train stations, the connections between them and about temporal notions.

- These data are organised in a KL-One-like **semantic network** implemented as a set of facts.

- The **semantic network** allows for expressing notions of class, sub-class, and instances.
- It also defines a notion of relation between classes, which applies to the different class instances.

Example 9.28

For a Directory application the semantic network involves the following classes:

- **People** (whose instances are the set of people known in the directory);
 - **Job** (whose instances are the people jobs).
- These two classes are related by the relation **JobOf**.

Example 9.29

To indicate that John is a lawyer, the semantic network involves the fact *JobOf(John, Lawyer)*.

- The **rational unit** can access the **semantic network** at any time of its inference procedure when the derived consequences depend on the nature of the data.

Example 9.30

In the Directory application, if the user asks for the job of John, the answer of **ARTIMIS** will depend on its query to the semantic network.

- The **semantic network** also involves semantic proximity notions, which are particularly useful to produce the cooperative reactions of an **ARTIMIS** agent.

Example 9.31

Lawyer will be considered as semantically closer (according to a certain metrics) to *Attorney* than to *Medical Doctor*.

- This construction enables to achieve two **symmetric operations**:
 - **Constraint relaxation**;
 - **Constraint restriction**.

- **Constraint relaxation** consists in providing a close answer to what has been requested when the “**exact**” response does not exist.
- The **rational unit** has access to the database through a procedural attachment engine that seeks satisfactory “**approximate**” solutions when no “**exact**” solution to the user’s request is found.
- This is implemented by “**compiling**” the **semantic network** into a product metric space where the dimensions are the application relations.
- A **distance function** $d(a, b)$ is assigned to each dimension in order to quantify the approximation made when relaxing a into b .
- **Constraint relaxation** is viewed as the operation of finding the nearest neighbours in the metric space.

Example 9.32

If we ask the Directory application to provide the list of the product managers, and it does not exist any, the inference procedure can trigger a constraint relaxation process in order to suggest, for example, the list of project managers.

- Conversely, the restriction process consists in finding a way to reduce a too wide set of solutions, by introducing **additional constraints** when the request is too **“weak”**.
- According to the current context, the constraint to be instantiated (by the user) could be, for example, the one corresponding to the dimension with the longest diameter.

Example 9.33

If there are fifty project managers in the Directory, restriction will provide the most discriminating dimension (for example, the department they work for), in order to pose a relevant question to better qualify the user's request.

Language processing unit

Input:

- This component relies on **semantic and linguistic models** to contextually interpret the user's natural language utterances, possibly combined with other communication media, and builds up a logical representation directly usable by the **rational unit**.

Output:

- This component achieves the converse function, that of generation, in order to express in a **“high level language”** the system responses.

Language understanding

- The goal of the understanding process is to reconstruct, in a **logical form**, the **dialogue acts** realised by the **input utterances**.
- The **utterance analysis** is based on detecting “**small**” **syntactic structures** that are going to activate one or more notions mainly coming from the **semantic network**.

- The **relation between** the **user's input vocabulary** and the **semantic network** is done by means of a **concept activation mechanism**.
- The concepts correspond to the semantic notions conveyed by the word sequences of the vocabulary.
- These activated notions partially depend on the application, but they also represent more general concepts such as:
 - User's intentions and beliefs;
 - Negation;
 - Existential and universal quantification;
 - Cardinalities;
 - etc.

- The **interpretation module** has at its disposal a set of activated concepts.
- It transforms them, through a semantic completion process, into a **well-formed logical sentence** that represents the semantic content of the dialogue act.
- This process relies on the assumption of semantic connectivity of user's utterance, that is to say that the concepts mentioned by the user have necessarily something to do with each others.
- It assumes that the utterance corresponds to a path in the **semantic network**.
- The **semantic completion process** aims at connecting to each others activated concepts through relations existing in the semantic network, even by adding, if necessary, concepts that were not explicitly mentioned.
- This process determines the understood notions in the utterance.

- The **interpretation module** has also to take into account the context of the user's utterance.
- It disposes of the concepts previously evoked both by the user and the **ARTIMIS** agent.
- A part of theses concepts can be used by the completion process.

- The **analysis method** (semantic-island driven analysis) and the **semantic completion** ensure certain **robustness** to the analysis/interpretation process, particularly required in the context of spontaneous speech.
- The only part that depends on user's language is the link between the used vocabulary and the semantic network concepts.
- The semantic data of the network represent language-independent notions.
- This makes particularly easy the application transfer from a (natural) language to another one.

Example 9.34 (Dialogue act)

Consider utterance:

“I’d like to know the phone number of a project manager in agent technology located at Lannion”, recognised by the speech recognition system as “I’d like X project manager agent technology X Lannion”), which in turn, activates the concepts of user’s intention, project manager, and Lannion.

The inferred semantic complements are that the request deals with a telephone number of an individual, whose position is project manager, on a topic that is agent technology, and whose location is Lannion.

Example 9.35 (Dialogue act (Formal))

Formally, this leads to the construction of the following dialogue act:

$$\langle u, \text{Inform}(s, I_u \text{Bref}_u \wedge x(\text{phone} - \text{number}(x) \wedge \exists y \text{ individual}(y) \\ \wedge \text{position}(y, \text{project} - \text{manager}) \wedge \text{topic}(y, \text{agent} - \text{technology}) \\ \wedge \text{location}(y, \text{lannion}) \wedge \text{phone} - \text{number}(x, y))) \rangle$$

meaning that the user (u) informs the system (s) that he wants ($I_u \dots$) to know ($\text{Bref}_u \dots$) the telephone number of a project manager in agent technology at Lannion.

Language generation

- The **generation module** achieves the converse task of the interpretation one.
- It must transcribe a **sequence of communicative acts** produced by the **rational unit** into an output understandable by the user.

The **generation process** proceeds in two phases:

① Determine the **surface acts**, the **utterances modes**:

- Declarative;
- Imperative;
- Interrogative.

and **reference acts**, the **designation modes**:

- nominal groups,
- pronouns,
- proper nouns,
- etc.

that achieve the **dialogue act(s)** to be sent to the user.

② Find the **best formulation** of the acts specified in the first phase, depending on the **linguistic resources actually available** and on the **current linguistic context**.

Example 9.36 (Language generation)

- If the system wants to inform the user that there is a relation of type **“department”** between **“the agent technology project manager”** and **“information technology”**, it would send the following message to the generator:

$$\langle s, \text{Inform}(u, \text{department}(\iota y(\text{individual}(y) \\ \wedge \text{position}(y, \text{project} - \text{manager}) \wedge \text{topic}(y, \text{agent} - \text{technology}) \\ \wedge \text{location}(y, \text{lannion}), \text{information} - \text{technology})) \rangle$$

Example 9.37 (Language generation (continued))

- According to the dialogue context, the generator will produce a declarative sentence with a proper noun and a nominal group, or a positive answer with a pronoun and a proper noun, or an elliptic sentence with a proper noun, etc.:
 - “The department of the project manager on agent technology located at Lannion is information technology”.
 - “Yes, she/he belongs to the information technology department”.
 - “Information technology department”.
 - etc.

Media processing layer

- This component allows for input and output formatting according to the targeted media.
- It is possible to connect an ARTIMIS-based application to any interface type, with almost no new code development.
- **ARTIMIS** agents already have interfaces to handle:
 - VXML and SSML (for voice platforms);
 - HTML (to be operated behind an HTTP server);
 - ACL (to interoperate with other FIPA compliant agents);
 - XML;
 - and also the possibility of direct TCP/IP connection.

Example 9.38

In an end-user application over the telephone, in order to have a voice interaction with an **ARTIMIS** agent, it is required to connect it to speech recognition and synthesis systems.

Some implementation features

- **ARTIMIS** is based on an explicit first-order modal reasoning process.
- Its communicative and cooperative behaviour is specified declaratively, in terms of logical properties, which are implemented as such.
- To change its behaviour only requires modifying this set of logical properties.
- No translation from logics to another language is needed.

- ARTIMIS technology is available on Solaris, Windows, and Linux platforms.
- A classical machine configuration (PC, 1 GHz, 256 MO Ram) is sufficient to simultaneously run about 30 ARTIMIS agents in real time.
- The ARTIMIS operational platform is compliant with Patrol-like supervising software.

- As constituents of ARTIMIS technology, communicative acts and cooperation principles enable ARTIMIS agents to be naturally part of multi-agent systems.
- To interact with other agents, ARTIMIS uses the FIPA ACL (Agent Communication Language) standard, and its associated content language, SL (Semantic Language).
- In fact, both of these languages come from ARCOL, the ARTIMIS communication language.

- **ARCOL** is a performative-based language, with a non-ambiguous, fine-grained semantics.
- An **ARCOL** message is specified as a CA type applied to a semantic content.
- The semantic contents of **CAs** may be of three different types:
 - Propositions;
 - Individuals;
 - Actions (including **CAs**).
- **ARCOL** introduces a set of primitive **CAs** and provides the formal mechanisms that enable to build macro-acts and complex interaction protocols.

Definition 9.39

Simplified models of **ARCOL** primitive acts:

- $\langle i, \text{INFORM}(j, \phi) \rangle$

meaning: agent i informs agent j that proposition ϕ is true

$$\text{FP} : B_i \phi \wedge \neg B_i (B_i f_j \phi \vee U i f_j \phi)$$

$$\text{RE} : B_j \phi$$

- $\langle i, \text{REQUEST}(j, a) \rangle$

meaning: agent i requests agent j to perform action (e.g., **CA**) a

$$\text{FP} : \text{FP}(a)[i \setminus j] \wedge B_i \text{Agent}(j, a) \wedge \neg B_i I_j \text{Done}(a)$$

$$\text{RE} : \text{Done}(a)$$

Definition 9.40

$\mathbf{FP}(a)$ being the feasibility preconditions of a , and $\mathbf{FP}(a)[i \setminus j]$ the part of the \mathbf{FP} s of a that are mental attitudes of i .

- $\langle i, \mathbf{CONFIRM}(j, \phi) \rangle$

meaning: agent i confirms to agent j that proposition ϕ is true

$\mathbf{FP} : B_i \phi \wedge B_i U_j \phi$

$\mathbf{RE} : B_j \phi$

- $\langle i, \mathbf{DISCONFIRM}(j, \phi) \rangle$

meaning: agent i disconfirms to agent j that proposition ϕ is true

$\mathbf{FP} : B_i \neg \phi \wedge B_i (U_j \phi \vee B_j \phi)$

$\mathbf{RE} : B_j \neg \phi$

Macro-acts

- Some **macro-acts** result from the **planning of a nondeterministic choice**.
- They are **selected by an agent** when it intends to achieve the **rational effect** of one of the acts composing the choice, no matter which one it is.
- To do that, one of the **feasibility preconditions** of the acts must be satisfied, no matter which one it is.
- **Macro-acts** cannot be achieved as such but are required to abstract.

Example 9.41

An *Inform* within the scope of a *Request*, to form a yn-question or a wh-question.

Definition 9.42 (Macro-acts for *Inform* acts)

In the case of *Inform* acts, such **macro-acts** are defined as follows:

- $\langle i, \text{INFORMIF}(j, \phi) \rangle$
 $\equiv \langle i, \text{INFORM}(j, \phi) \rangle \mid \langle i, \text{INFORM}(j, \neg\phi) \rangle$

meaning: agent i informs agent j that proposition ϕ is true or that it is false

$$FP : Bif_j\phi \wedge \neg B_i(Bif_j\phi \vee Uif_j\phi)$$

$$RE : Bif_j\phi$$

Definition 9.43 (Macro-acts for *Inform* acts (continued))

In the case of *Inform* acts, such macro-acts are defined as follows:

- $\langle i, \text{INFORMREF}(j, \iota x \delta(x)) \rangle$
 $\equiv \langle i, \text{INFORMREF}(j, \iota x \delta(x) = r_1) \rangle \mid \dots \mid$
 $\langle i, \text{INFORMREF}(j, \iota x \delta(x) = r_k) \rangle$

meaning: agent i informs agent j of the value of the referent x denoted by $\delta(x)$

$$FP : \text{Bref}_i \delta(x) \wedge \neg B_i(\text{Bref}_j \delta(x) \vee \text{Uref}_j \delta(x))$$

$$RE : \text{Bref}_j \delta(x)$$

where $\iota x \delta(x)$ is a definite description and r_k identifying referring expressions.

- On this basis, using the **rationality principles**, a set of **dialogue acts**, that are, in fact, **inter-agent communicative plans** are specified.

Example 9.44

The following model for a **direct strict yes-no-question plan** can be build up:

$$\begin{aligned} &\langle i, YN - QUESTION(j, \phi) \rangle \\ &FP : B_i Bif_j \phi \wedge \neg Bif_i \phi \wedge \neg Uif_i \phi \wedge B_i \neg B_j (Bif_i \phi \vee Uif_i \phi) \\ &RE : Bif_i \phi \end{aligned}$$

Such an act encapsulates the following two-act plan:

$$\langle i, REQUEST(j, \langle j, INFORMIF(i, \phi) \rangle) \rangle; \langle j, INFORMIF(i, \phi) \rangle$$

Definition 9.45

Let us call:

- **SL** the **first-order modal language** in which the rational interaction theory underlying **ARTIMIS** is couched;
 - **SCL** the **semantic content language of communicative acts**.
-
- If it is taken to be **SL** itself, **SCL** happens to be a **very expressive language** but may turn out to be **complex** to use as such if an **ARTIMIS**-like agent technology is not available to process it.
 - For “**simpler**” agents and for **implementation reasons**, the **content language** may be taken to be a **less-expressive formalism**, *e.g.*, a subset of **SL**.

Definition 9.46

Suppose that one want to **simplify SCL**, for example by **restricting it to a first-order predicate logic language**:

- Let us call this simplified version **SCL1**.
 - **ARCOL** is accordingly restricted to a language **ARCOL1**.
-
- We would like to enable the agents to **communicate** their **mental attitudes** (beliefs, intentions, etc.).
 - Augment the set of **ARCOL1**'s communicative acts with complex acts (typically, macro-acts) that intrinsically integrate in their **semantics relevant mental attitudes**.

Example 9.47

- The act of agent **i informing** agent j that it (i.e., i) has a given intention, has the following semantics in **ARCOL**:

$$\begin{aligned} \langle i, \text{INFORM}(j, l_i p) \rangle: \\ \text{FP} : l_i p \wedge \neg B_i(B_i f_j l_i p \vee U i f_j l_i p) \\ \text{ER} : B_j l_i p \end{aligned}$$

- This act can be defined in **ARCOL1** as follows:

$$\begin{aligned} \langle i, \text{INFORM-I}(j, p') \rangle: \\ \text{FP} : l_i p' \wedge \neg B_i(B_i f_j l_i p' \vee U i f_j l_i p') \\ \text{ER} : B_j l_i p' \end{aligned}$$

- This **ARCOL1**'s new act $\langle i, \text{INFORM-I}(j, p') \rangle$ is only an **abbreviated** form of $\langle i, \text{INFORM}(j, l_i p) \rangle$ from **ARCOL** since propositions p and p' are expressed in **SCL** and **SCL1**, respectively.

- The **major downside** is that it requires the **definition of an additional communicative act** each time the **semantic content** cannot be expressed in the **first-order modal language**.
- Since agents are intended to express communicative acts that refer to **mental attitudes**, the corresponding technology must enable the manipulation of such notions.
- If agents have these notions explicitly represented in their own kernels, there is no **a priori** reason not to include them into the agent communication language.

A natural human-computer dialogue agent

- **ARTIMIS** has been demonstrated on several service prototypes displaying a natural dialogue with a human user, and on different interaction media and languages.
- It has also been experimented in the context of different field trials in various application domains:
 - Directories;
 - Tourist guides;
 - Banking and stock exchange;
 - Travel;
 - Internet portals;
 - etc.

- **ARTIMIS** has been commercially deployed by France Telecom in a FAQ-like application, namely the help of Wanadoo instant messaging service, **Le Messenger**.
- This application, called **Assistant Messenger**, consists in a conversational buddy that assists the user in using the instant messaging.
- **ARTIMIS** is able to manage an intelligent cooperative dialogue in natural language.

Example 9.48 (Plan Resto : Paris restaurant guide)

The following example presents a sample spoken dialogue collected over the telephone:

- S:** Welcome to PlanResto, Paris restaurant guide with direct connection. The general user guide can be activated at any time by saying general user guide. If you are not a regular user please say I am not a regular user. May I help you?
- U:** I'm not a regular user.
- S:** PlanResto is a Paris restaurant guide. As with a human operator, converse with it in a natural way. Do not hesitate to correct it in case of misunderstanding. Feel free to interrupt it. Speak loud enough and distinctly, with no pausing and no excessive articulating. Avoid noisy surrounding. Can I help you formulating your request?

Example 9.49 (Plan Resto : Paris restaurant guide (continued))

U: I'd like a vegetarian restaurant close to the Picasso Museum.

S: There isn't any vegetarian restaurant close to the Musée Picasso. However, there are vegetarian restaurants close to the Musée Cognacq-Jay, close to Filles du Calvaire and close to Hôtel Carnavelet. Which suggestion are you interested in?

U: I'd like the last solution.

S: I found two vegetarian restaurants close to the Hôtel Carnavalet. The first reference is the restaurant Le Loir dans la Théière. Do you want to have more information, hear about the next restaurant or make a new request?

Example 9.50 (Plan Resto : Paris restaurant guide (continued))

U: What's its address?

S: The restaurant Le Loir dans la Théière is located 3 rue des Rosiers. Do you want more information, to be connected to this restaurant or make a new request?

U: Finally, I would prefer a Canadian restaurant.

S: There isn't any Canadian restaurant close to Hôtel Carnavalet. However, there are Canadian restaurants close to the Cloître des Billettes. Does this suggestion suit you?

U: Yes please.

Example 9.51 (Plan Resto : Paris restaurant guide (continued))

S: You want a Canadian restaurant close to the Cloître des Billettes. I've found one called Equinox. Do you want more information, to be connected with this restaurant or make another request?

U: Could you give me the phone number of this restaurant?

S: The phone number of Equinox restaurant is 01 42 71 92 41.

U: It's OK. Goodbye.

S: Do you really want to leave Plan Resto?

U: Yes.

S: Thanks for calling Plan Resto. Goodbye for now!

ARTIMIS help buddy for the Wanadoo instant messaging: L'Assistant Messenger

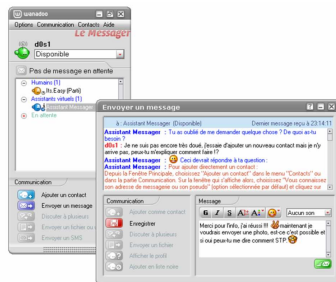


Figure 76: Chat with a virtual contact: Wanadoo "Assistant Messenger"

L'Assistant Messenger

Example 9.52 (L'Assistant Messenger)

Example translated from French.

- S:** Welcome Droopy! I hope you're well. I'm L'Assistant-Messenger. I'm here to explain to you how to use Le-Messenger-Wanadoo. You can converse with me. I'll try to help you. What can I do for you?
- U:** Hi, it's the first time I am using Le-Messenger. Could you tell me what is it and what availability means?
- S:** Ok, so you want information about "instant messaging" and "availability". I need more precisions. I can suggest some help:
1) to discover Le-Messenger-Wanadoo 2) to know what an availability state is Which suggestion are you interested in?

Example 9.53 (L'Assistant Messenger (continued))

U: Give me 1.

S: Here is the answer I found: Le-Messenger Wanadoo is an Internet real time communication tool. The connection state of each subscriber is memorised in a central server. Thus, at any time, it is possible to know if your contacts are connected and available to communicate. To use it, you need an email address and a password. The pseudo can only be created on the Wanadoo portal. I can give you additional information, if this answer is not sufficient for you. Do you want it?

U: No thank you, it's ok for now.

[Later...]

Example 9.54 (L'Assistant Messenger (continued))

- S:** So, you're back! Do you realise that it's more than 1 day that we haven't talked. Do you have any new question on Le-Messenger-Wanadoo? I'm always available to help you. What can I do for you?
- U:** Could you tell me how to delete a contact?
- S:** I've got the answer to your question: To take off a person from your contacts, select your addressee list, then you have two ways to delete the contact: - Using the contextual menu: right-click and choose "Delete". - Using the menu "Contacts": choose entry "Contacts" then select entry "Delete a contact or a file". What can I do for you now?

Example 9.55 (L'Assistant Messenger (continued))

U: What's your name?

S: My name is HelpWare, L'Assistant-Messenger. I'm just here to give you any information you need about Le-Messenger-Wanadoo. What do you want to know?

U: Do you know any joke?

S: I've 3 heads, 3 legs, 1 arm and 6 fingers, who am I? A liar!

U: Very funny, how old are you? :-)

S: Right now I am precisely 10 months, 25 days, 8 hours, 29 minutes and 40 seconds!

Example 9.56 (L'Assistant Messenger (continued))

U: Are you a genius or a machine!?

S: A human or a machine, no matter what I am :-) I'm here to help you use Le-Messenger-Wanadoo. How can I help you?

U: It's ok, thanks! See you.

S: If you don't need me anymore, you can just close my dialogue window!

ARTIMIS as mediation agent

- **ARTIMIS** is virtually located between a human user and domain-specific agents.
- By interacting with such agents, it acquires from them pieces of information that it aggregates and organises before delivering them as a global solution to the user in a high level language and a user-friendly and cooperative format.
- **ARTIMIS** has played this role as a **Travel Assistant**, in particular in the context of the *Agentcities* IST European.
- It delivers to the user information on travelling between the main European cities.
- It gets such information by interacting with train and flight schedule information agents.

ARTIMIS displays an added-value as a **mediation agent** on several aspects:

- Offering a global trip by aggregating the different responses from the information agents.
- Suggesting the most interesting journey according to different criteria (duration, earliest arrival, latest departure, etc.).
- Suggesting alternative travels when the one targeted by the user does not exist.
- Interacting with the user in natural dialogue.
- etc.

- In the Travel Assistant application, ARTIMIS uses a FIPA-Query standard protocol to interact with the domain-specific agents.
- In this protocol, the first communicative act sent by ARTIMIS to the information agents is a Query-ref or a Query-if.
- Each addressee responds either by sending a Refuse, or by sending an Inform conveying the information found out.
- The mediation agent then organises the obtained information according to the user's request.

- It is worth noting that although **ARTIMIS** can use a **FIPA standard protocol** to interact with simpler agents.
- It is also able to process “**free-style**” requests coming from the user or from another agent, which do not fit a specific protocol.

Example 9.57

For instance, to identify flights according to some criteria, the interlocutor could ask **ARTIMIS** for the sought information using a **Query-ref** act (“What are the flights that are so and so?”), or an **Inform** act of the intention to know such flights (“I want to know the flights that are so and so.”).

- In the second case, the semantic content of the act expresses an intention of the interlocutor.
- **ARTIMIS** handles such a content and, on the basis of its cooperation principles comes to the same behaviour as for a regular **Query-ref**.

Example 9.58 (Travel Assistant)

S: Welcome on ARTIMIS-Voyage. How can I help you?

U: I'd like to go to Nice on tomorrow.

S: A trip to Nice on Friday, April 1st, 2005. What's your departure city?

U: I'd like to leave from Brest in the afternoon and arrive before 6pm.

S: A trip from Brest to Nice on Friday, April 1st, 2005, leaving after 12am and arriving before 6pm. How do you want to travel, by train or by plane?

U: I don't care.

S: A trip by train or by plane from Brest to Nice on Friday, April 1st, 2005 leaving after 12am and arriving before 6pm. Do you want to wait for the response, to receive it by email, or to be called back?

Example 9.59 (Travel Assistant (continued))

U: I am waiting.

S: Thank you for waiting. I am going to consult the travel sites.

[...]

There is a flight leaving from Brest on Friday, April 1st, 2005 at 2:15pm and arriving at Nice at 5:35pm. Would you like the details of this journey, the alternative solutions, or any other information?

U: Give me the details.

S: There are two steps. First step: departure from Brest by plane on Friday, April 1st, 2005 at 2:15pm; arrival at Paris Orly-West at 3:30pm. Second step: departure from Paris Orly-West by plane at 4:10pm; arrival at Nice at 5:35pm. Would you like the alternative solutions?

Example 9.60 (Travel Assistant (continued))

U: Yes, the alternatives please.

S: The quickest trip by plane is: departure from Brest on Friday, April 1st, 2005 at 5:40pm, arrival at Nice at 8:50pm. Would you like to know the steps of this journey?

U: And by train?

S: I don't know any trip by train from Brest to Nice on Friday, April 1st, 2005 leaving after 12pm and arriving before 6pm. However, there is a train departing from Brest on Friday the 1st of April 2005 at 8:49am, and arriving at Nice at 10:51pm. Would you like to know the steps of this journey or an alternative solution?

U: I'd like to have the details.

Example 9.61 (Travel Assistant (continued))

S: There are 3 steps. First step: Departure from Brest by train on Friday, April 1st, 2005 at 8:49am; arrival at Paris Montparnasse at 1:30pm. Second step: departure from Paris Gare de Lyon by train at 2:21pm; arrival at Toulon at 7:34pm. Third step: departure from Toulon at 9:11pm; arrival at Nice on at 10:51pm. Would you like any other information?

U: No, thank you.

S: Do you really want to leave ARTIMIS-Voyage?

U: Yes please.

S: Thanks for calling ARTIMIS-Voyage. Goodbye for now!

References I



P. Bretier and Sadek D.

A rational agent as the kernel of a cooperative spoken dialogue system:
Implementing a logical theory of interaction.

*In Proceedings of the ECAI'96 workshop on Agent Theories,
Architectures, and Languages (ATAL), Budapest, Hungary, 1996.*

In Intentions in Communication. MIT press, 1990.



Sadek D.

Communication theory = rationality principles + communicative act
models.

*In Proceedings of the AAAI'94 Workshop on Planning for Interagent
Communication, Seattle, WA, 1994.*

References II



Sadek D., A. Ferrieux, A. Cozannet, P. Bretier, F. Panaget, and J. Simonin.
Effective human-computer cooperative spoken dialogue: The ags demonstrator.

In Proceedings of ICSLP'96, Philadelphia, 1996.



G.W. Garson.

Quantification in modal logic.

In Handbook of philosophical logic, Volume II: Extensions of classical Logic, pages 249–307. D. Reidel Publishing Company, 1984.



J.Y. Halpern and Moses Y.

A guide to the modal logics of knowledge and belief: a preliminary draft.

In Proceedings of the IJCAI'85, Los Angeles, CA, 1985.



Doyle J.

Rationality and its role in reasoning.

In Proceedings of AAAI'90, Boston, MA, 1990.

References III



K. Konolige.

A First-Order Formalisation of Knowledge and Action.

Hayes, 1982.



S Russel.

Rationality and intelligence.

In *Proceedings of IJCAI'95*, Montreal, 1995.



D. Sadek.

Dialogs acts are rational plans, 2000.



D. Sadek and H Brouchoud.

Intelligent agents.

In *Networks: Internet, telephony, multimedia - Convergences and complementarities*. De Boeck University, 2002.

References IV



Martinez T., Bretier P., and Louis V.

Artimis : assistant de voyage et médiateur intelligent dans un système multi-agent.

In *Proceedings of JFSMA'03*, Tunisia, 2003.