CAP - Concurrent Action and Planning:
Using PVM-Prolog to Implement Vivid Agents

M. Schroeder¹, R. Marques², G. Wagner³, J. C. Cunha²

November 28, 1996

Abstract

First, we describe PVM-Prolog, a Prolog core extended by an interface to PVM, the Parallel Virtual Machine, a standard software which allows to view a network of heterogeneous machines as a single parallel computer. Besides PVM's coarse-grain parallelism, PVM-Prolog includes a process-internal thread concept to realize fine-grain concurrency.

Second, we review the concept of vivid agents [Wag96] and develop the architecture CAP (Concurrent Action and Planning) that serves as an operational semantics for vivid agents.

Finally, we merge the above two lines of research by showing that PVM-Prolog is an excellent candidate to implement vivid agents and multi-agent systems, in general: the coarse-grain parallelism is used to spawn agents in a network, while the fine-grain concurrency is used to run a perception-reaction-cycle and a planning facility for each agent concurrently.

1 Introduction

In the past few years research in multi-agent systems (MAS) developed tremendously. However, up to now there is no standard MAS programming language available. In this paper, we take a first step towards such a MAS programming language by showing that PVM-Prolog is an excellent basis to implement multi-agent interpreters.

A state-of-the-art programming language for multi-agent systems has to meet the following requirements:

- Reactive and Pro-active Behaviour. The language has to support both reactive and pro-active behaviour. I.e. on the one hand the agent...
has to react timely to incoming events and on the other hand it has to pursue long-term goals based on some planning facilities.

- **Formal Semantics.** Multi-agent applications are complex and therefore likely to be error-prone. Since many applications require high safety standards the language needs a formal semantics in order to simulate and verify the behaviour of the programs.

- **Executable Specifications.** The specification of agent behaviour has to be declarative and executable.

- **Hardware Independence.** Applications are usually developed on a different hardware than they are later run on. Furthermore, hardware may be updated and is subject to change. Therefore the language must be portable.

- **Openness.** The language has to provide facilities to create new agents at run-time and it has to be possible for agents which work independent of the multi-agent system to join the system.

- **Heterogeneity.** The language has to support the use of agents based on different concepts and architectures and implemented in different programming languages.

- **Modularity.** The agent architecture underlying the language has to be modular, so that different types of agents can be composed. For example, a diagnosis agent [SaAMP96, SW96] needs a powerful knowledge base and few planning, whereas a scheduling agent may need an optimal planner and a relational database only.

The paper is divided into three parts. In the first part we give an overview over the technical concepts and primitives of PVM-Prolog [MC95a, CM96]. In the second part we review the concept of vivid agents [Wag96] and develop an operational semantics and the CAP architecture (Concurrent Action and Planning) for vivid agents. In the third part we briefly describe the implementation of the CAP architecture in PVM-Prolog. We round out the picture with an example of a MAS specification and its execution. We conclude by discussing which of the above requirements are and which can be met in principle by a multi-agent language implemented in PVM-Prolog.

2 PVM-Prolog

PVM-Prolog extends Prolog with extra-logical mechanisms to support higher level parallel and distributed logic programming. PVM-Prolog takes PVM [Gen94], a well known parallel programming system for distributed memory systems as its supporting system. PVM-Prolog integrates PVM's distributed memory programming model and extends it with mechanisms for multi-threading a
Prolog process. These are important to make it easier to model distributed abstractions relying on internal concurrency of its processing entities.

In this section we briefly review PVM-Prolog’s programming model and its current implementation. For a more detailed description of PVM-Prolog see [MC95a, CM96].

2.1 The Multi-threaded Prolog Process

In our model, a Prolog process is a place for the concurrent execution of a set of threads, which can be seen as evaluating concurrent queries to the Prolog database. Goals resolved by different threads are independent, in the sense that they are not allowed to share variables.

<table>
<thead>
<tr>
<th>Thread Control</th>
<th>Term Queues</th>
<th>Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_create(+Goal, +Stacks, +Priority, +Activity, -ThreadId)</td>
<td>q_create(+QName, +MaxTerms, +MaxMem, -QId)</td>
<td>t_activity(+ThreadId, +ThreadIdActivity)</td>
</tr>
<tr>
<td>t_mymtid(-ThreadId)</td>
<td>q_name(?QName, ?QId)</td>
<td>t_priority(+ThreadId, +ThreadIdPriority)</td>
</tr>
<tr>
<td>t_kill(+ThreadId)</td>
<td>q_put[+QId, +Term]</td>
<td></td>
</tr>
<tr>
<td>q_get[+QId, -Term]</td>
<td>q_select[+QId, -Term, -Queue]</td>
<td></td>
</tr>
<tr>
<td>q_destroy(+QId)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Main multi-threading predicates
Thread control predicates (see table 1\(^1\)) include t\_create, which creates a thread to solve the specified Goal and t\_kill which terminates a thread’s execution.

Thread scheduling is done in a preemptive round-robin scheme, parametrized by two parameters, activity and priority. Higher priority ready threads exclude lower-priority ones from running. Higher activity threads have a higher execution quantum, so they get more processor time than lower activity ones. t\_priority and t\_activity allow the programmer to control these parameters.

Term queues have maximum capacities for memory and number of terms, defined at thread creation. Operations which would exceed that capacity or would require a term from an empty queue cause the accessing thread to block.

Term queue primitive operations include get and put, as well as a non-deterministic select predicate which allows the receiving thread to fetch a term from one of a number of queues. All these operations have non-blocking (in which they fail instead of blocking) and priority versions (in which their priority is used to determine the order in which concurrent requests will be satisfied).

### 2.2 The PVM Prolog Interface

The Prolog programmer can access the PVM environment through PVM interface predicates which are the Prolog counterpart to the PVM library routines. They are mostly deterministic, relying on a procedural interpretation of Prolog and on appropriate side-effects.

In table 2 we can see an outline of such predicates. Most of them, such as pvm\_exit, pvm\_parent and most of the grouping predicates, just mimic their C language interface counterparts. See [Gea94] for a complete reference on PVM routines.

Others, such as pvm\_config, pvm\_tasks, the options and machine configuration predicates, translate integer constants and arrays into Prolog atoms and lists (see [MC95a, CM96] for a complete reference). Structures are converted into Prolog compound terms with a functor with the same name as the ‘C’ structure type.

Routines that return a success or failure status (such as pvm\_norecv) make use of Prolog non-determinism, succeeding or failing, according to the result of the operation.

Packing Prolog terms into PVM messages is implicit in the communication predicates. As Prolog terms have incorporated typing information it’s not necessary to burden the programmer with this task. This leads to representing data structures (Prolog terms) in a Prolog-like way. To overcome difficulties of communication between Prolog tasks and PVM tasks written in other languages a PVM-Prolog0 interface as been defined [CM96] which offers a more direct correspondence between with the PVM calls.

\(^1\)In this and other tables we use the convention of \(+\) for input arguments, \(-\) for output and \(?\) for input/output. Square brackets \([]\) denote optional parts on predicate name syntax, for the sake of brevity.
Process Control
- `pvm_gettid(-TID)`
- `pvm_spawn(+LogicProgram, +Goal, +OptList, ...)`
- `pvm_exit`  

Information
- `pvm_parent(-TID)`
- `pvm_tidtohost(+TID, -DTID)`
- `pvm_config(-NHosts, -NArch, -HostInfoList)`
- `pvm_tasks(+Where, -N Tasks, -TaskInfoList)`  

Dynamic Configuration
- `pvm_addhosts(+HostList, -StatusList)`
- `pvm_delhosts(+HostList, -StatusList)`

Options
- `pvm_setoptions(+Option, +Value)`
- `pvm_getoptions(+Option, -Value)`  

Communication
- `pvm_send(+TID, +Tag, +Term)`
- `pvm_mcast(+TIDList, +Tag, +Term)`
- `pvm_mrecv(+TID, +Tag, -Term)`

Dynamic Groups
- `pvm_joingroup(+GName)`
- `pvm_lvgroup(+GName)`
- `pvm_gsize(+GName, -NElems)`
- `pvm_barrier(+GName, +Count)`
- `pvm_bcros(+GName, +MsgTag, +Term)`

Table 2: Main PVM interface predicates

<table>
<thead>
<tr>
<th>Prolog Predicate Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PVM-Prolog</strong></td>
</tr>
<tr>
<td>PVM</td>
</tr>
<tr>
<td>Operating System and Hardware Layers</td>
</tr>
</tbody>
</table>

Figure 2: PVM-Prolog Layered Architecture

2.3 Implementation Outline

Our system design tries to address the issues of execution on a non-homogeneous hardware system, including distinct Prolog systems. Evolution of the hardware/software platforms is expected and should bring as little modifications in the system as possible. Bearing this in mind a multi-layered architecture has been designed.

The PVM Interface layer supports the communication with other processes, and might evolve to other distributed systems (or simply newer PVM versions) as new technology arises. The Scheduler and Prolog Executor cooperate in order to achieve the multi-threaded execution of the Prolog program.

In the following a brief description is presented of the implementation issues brought by these components.
**Prolog Executor and the Scheduler**  Our approach to multi-threading consists in having user-level threads controlled by a scheduler. The scheduler is an independent component which interacts with the Prolog Executor by setting its execution context. The context which is to be given to the executor is determined by the scheduler's internal algorithms.

**PVM-Prolog Interfacing**  Interfacing PVM from Prolog brings problems based on the differences of type systems used and execution model (procedural PVM vs. logic Prolog).

At implementation level the former is solved by conversion functions that transform Prolog terms into C data types. The latter doesn't create problems as far as a single process is concerned, since Prolog also has a procedural interpretation. However in creating a new Prolog process the Prolog program and Prolog goal are treated as data of the Prolog Interpreter, instead of the direct mapping of PVM processes to executable programs assumed by PVM.

Encoding Prolog terms as PVM messages is another of the problems to solve. We use a prefix representation, following the structure of the Prolog term, instead of a textual representation generated by the Prolog I/O system. Atom and functor names are present in the message, so new tasks (with a possibly different internal atom naming) can join an already running system.

## 3 Vivid Agents

A *vivid agent* is a software-controlled system whose state is represented by a knowledge base, and whose behaviour is represented by means of *action* and *reaction rules*. Following [Sho03], the state of an agent is described in terms

![Diagram of interactions between the Scheduler and the Prolog Executor](image-url)
of mental qualities, such as beliefs and intentions. The basic functionality of a vivid agent comprises a knowledge system (including an update and an inference operation), and the capability to represent and perform actions in order to be able to generate and execute plans. Since a vivid agent is 'situated' in an environment with which it has to be able to communicate, it also needs the ability to react in response to perception events, and in response to communication events created by the communication acts of other agents.

Notice that the concept of vivid agents is based on the important distinction between action and reaction: actions are first planned and then executed in order to solve a task or to achieve a goal, while reactions are triggered by perception and communication events. Reactions may be immediate and independent from the current knowledge state of the agent but they may also depend on the result of deliberation. In any case, they are triggered by events which are not controlled by the agent.

We do not assume a fixed formal language and a fixed logical system for the knowledge-base of an agent. Rather, we believe that it is more appropriate to choose a suitable knowledge system for each agent individually according to its domain and its tasks.

3.1 Specification of Vivid Agents

A vivid agent comprises five main ingredients: a knowledge base with the agent's beliefs, an event queue, a goal queue, reaction rules that specify the reactive behaviour and action rules used to generate plans achieving the agents goals.

Definition 3.1 Agent
Let $L_{KB}$ denote the language of the knowledge base. An agent $A = \langle X, EQ, GQ, RR, AR \rangle$, on the basis of a knowledge system $K$ consists of

1. a knowledge base $X \in L_{KB}$,
2. an event queue $EQ$ being a list of instantiated event expressions,
3. a goal queue $GQ$ being a list of literals,
4. a set $RR$ of reaction rules, consisting of epistemic and physical reaction and interaction rules which code the reactive and communicative behaviour of the agent, and
5. a set $AR$ of action rules that allow the agent to plan how to achieve a goal.

A multi-agent system is a tuple of agents: $S = \langle A_1, \ldots, A_n \rangle$

3.1.1 Reaction Rules

Reaction rules encode the behaviour of vivid agents in response to perception events created by the agent's perception subsystems, and to communication
events created by communication acts of other agents. We distinguish between
epis-temic, physical and communicative reaction rules, and call the latter interaction rules. We use \( L_{PEvt} \) and \( L_{CEvt} \) to denote the perception and communication event languages, and \( L_{Evt} = L_{PEvt} \cup L_{CEvt} \). The following table describes the different formats of epistemic, physical and communicative reaction rules:

\[
\begin{align*}
\text{Epistemic Reaction:} & \quad \text{The agent’s knowledge base is updated.} \\
\text{Physical Reaction:} & \quad \text{do}(\alpha(V)) \text{ calls a procedure realizing the action } \alpha \text{ with parameters } V. \\
\text{Communicative Reaction:} & \quad \text{sendMsg}[\eta(V), R] \text{ sends the message } \eta \in L_{CEvt} \text{ with parameters } V \text{ to the receiver } R.
\end{align*}
\]

Both perception and communication events are represented by incoming messages. In general, reactions are based both on perception and on knowledge. Immediate reactions do not allow for deliberation. They are represented by rules with an empty epistemic premise, i.e. \( \text{Cond} = \text{true} \). Timely reactions can be achieved by guaranteeing fast response times for checking the precondition of a reaction rule. This will be the case, for instance, if the precondition can be checked by simple table look-up (such as in relational databases or fact bases).

Reaction rules are triggered by events. The agent interpreter continually checks the event queue of the agent. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfied in the current knowledge base, all free variables in the rules are instantiated accordingly resulting in a set of triggered actions with associated epistemic effects. All these actions are then executed, leading to physical actions and to sending messages to other agents, and their epistemic effects are assimilated into the current knowledge base.

3.1.2 Action Rules

In order to achieve a goal or to solve a task in a situation described by \( X_0 \in L_{KB} \), an agent generates a suitable plan \( P \) being a sequence of action rules
ar1, . . . , arn, such that when the corresponding sequence of actions is performed in X0, it leads to a situation P(X0) ∈ LKB where G holds:

\[ P = ar_n \circ \ldots \circ ar_1, \quad \text{and} \quad P(X_0) \vdash G \]

where P is applied to X0 as a composed function. Syntactically, action rules have the form

\[ \alpha(V) : \text{Eff} \leftarrow \text{Cond} \]

If \( \alpha \) is applied in a situation satisfying \text{Cond} the situation changes according to \text{Eff}.

Notice that our concept of planning on the basis of action rules in knowledge systems can be viewed as a generalization of the STRIPS paradigm which corresponds to planning on the basis of relational databases. Therefore, the frame problem is solved in the same way as in STRIPS: by means of a minimal change policy incorporated in the update operation of a knowledge system.

4 CAP – an Architecture for Concurrent Action and Planning

4.1 Motivation

An architecture to realize a vivid agent involves four main components: a knowledge base, a reactive component, a planner, and a plan execution facility. Since action and planning are two nearly independent tasks it is useful to separate them in order to achieve greater modularity and efficiency by running two processes for action and planning in parallel. We call this architecture which is depicted in Fig. 4 CAP for Concurrent Action (subsuming both reactions and planned actions) and Planning. The reactive component receives incoming messages representing perception and communication events and generates a reaction based on the set of reaction rules. The reactions depend on the agent’s knowledge base and may update it. The planner runs concurrently with the perception-reaction-cycle on a copy of the knowledge base. Therefore the plans depend on the knowledge base but do not change it. Once the planner has generated a plan it communicates it to the action component, where plan execution is interleaved with reaction.

Formally we can capture the CAP architecture by a tuple \((A, P)\) of an action component A and a planning component P:

**Definition 4.1 Agent State**

Let \( X_1 \) and \( X_2 \) be knowledge bases, \( EQ \) an event queue, \( PQ \) a plan queue, \( GQ \) a goal queue and \( \text{Flag} \in \{a, p\} \) a flag. Then \( A = (X_1, EQ, PQ, \text{Flag}) \) a action state, \( P = (X_2, GQ) \) planning state and \((A, P)\) agent state.

The different components of the agent state need some explanation.
First of all, there are two knowledge bases: The reaction state contains an up-to-date knowledge base $X_1$ and the planning state a knowledge base $X_2$ which is copied from $X_1$ once in a while. The knowledge base $X_1$ is subject to rapid change so that it would be inappropriate for planning. Thus, plans are generated with reference to a fixed copy of $X_1$. Of course, we expect that the changes of $X_1$ do normally not affect the quality of plans based on the older $X_2$. Thus the plan is executed with respect to $X_1$. In case any planned action is no more applicable, replanning is initiated.

**Event Queue**  The event queue represents the agent’s connection to the environment. The event queue stores incoming messages from the agent’s perception subsystems and from other agents. The stored events are consumed one by one triggering appropriate reactions.

**Plan Queue** The planner generates plans and communicates them to the reaction component which is in lieu of acting. The plans in the queue are executed step by step. If a plan’s next action is not applicable anymore the plan is removed from the queue and the failure is communicated to the planner so that the goal is replanned.
Flag  The flag indicates for the action state whether it has to react in response to events or whether a planned action is to be executed.

Goal Queue  The planning state comprises a goal queue that informs the planner of what goals are still to be achieved.

4.2 Transition Semantics

We can describe the temporal behaviour of an agent by means of transitions between agent states. Five kinds of transitions are performed: perception, reaction, plan execution, replanning and planning.

(Perception)  An incoming perception event message $\varepsilon$ is added to the event queue:

$((X_1, EQ, PQ, Flag), P) \xrightarrow{\varepsilon} ((X_1, EQ + \varepsilon, PQ, Flag), P)$

(Reaction)  With the flag set for reaction and an event $\varepsilon$ queued the transition

$((X_1, \varepsilon : EQ, PQ, r), P) \xrightarrow{RR} ((RR_\varepsilon(X_1), EQ, PQ, p), P)$

is performed, i.e. $\varepsilon$ is removed from the event queue, according to the reaction rules a reaction is performed leading to an update of the knowledge base, and the flag is switched to plan execution.

If the flag is set for reaction but there are no events in the event queue then plan execution is enabled by flipping the flag:

$((X_1, [], PQ, r), P) \rightarrow ((X_1, [], PQ, p), P)$

(Plan Execution)  If the flag is set for plan execution and the next planned action $\pi$ is executable (i.e. $\pi : Eff \leftarrow Cond$ and $X_1 \vdash Cond$) then it is removed from the plan queue and the flag is set for reaction:

$((X_1, EQ, (G, \pi : \Pi) : PQ, p), P) \xrightarrow{\pi} ((RR_\varepsilon(X_1), EQ, (G, \Pi) : PQ, r), P)$

if $\pi : Eff \leftarrow Cond$ and $X_1 \vdash Cond$.

If the flag is set for plan execution but there is no action left in the plan queue then reaction is enabled by flipping the flag:

$((X_1, EQ, [], p), P) \rightarrow (X_1, EQ, [], r), P)$

We assume that the planned actions can be seen as atomic units which are not decomposable anymore. It may be desirable to interleave reaction and action with a different priority scheme.
(Replanning) If a planned action is not executable anymore the plan is removed from the plan queue, the goal is added to the goal queue and the planner knowledge base is updated:

\[ ((X_1, EQ, (G, \pi : \Pi) : PQ, p), (X_2, GQ)) \xrightarrow{\pi} ((X_1, EQ, PQ, r), (X_1, GQ+G)) \]

if for all action rules \( \pi : Eff \leftarrow Cond \) and \( X_1 \not\models Cond \).

An alternative to removing the full plan from the plan queue is to remove only the action and initiate planning for the condition \( Cond \). But since the whole plan \( \Pi \) is based on the planners old copy of the knowledge base it may be the case that \( \Pi \) is not very useful anymore.

(Planning) Independent from the action state the planner generates plans for the goals and communicates them to the reaction state:

\[ ((X_1, EQ, PQ, F), (X_2, G : GQ)) \xrightarrow{planG} ((X_1, EQ, PQ+(G, \Pi, F)), (X_2, GQ)) \]

if \( \Pi(X_2) \models G \).

The above definition has several advantages over previous approaches: Most important, (re)acting and planning are completely separated tasks. In many other approaches control is distributed among the reactive and the planning component in a fixed scheme which gives rise to the problem of guaranteeing effective response times by means of bounded (or real-time) rationality. In our CAP architecture (re)actions and planning are performed concurrently and can communicate asynchronously through the plan and goal queues. This may solve the problem of bounded rationality since we can tune the responsiveness of agents by assigning different priorities to these tasks.

5 Implementation

The CAP architecture is easily implementable in PVM-Prolog. In fact, it was motivated by practical experience. Given a multi-agent system represented as a tuple \((A_1, \ldots, A_n)\) of agents all \(A_i\) run as parallel processes on distinct machines. Besides this coarse-grain parallelism, a particular agent \(A = (A, P)\) installed on a single machine invokes two parallel tasks or threads to run \(A\) and \(P\). In our implementation the MAS is initially set up by configuring the network and spawning the agents in the network using the primitive \texttt{pvm_spawn} (see the initial process in figure 5). To set up a particular agent the primitive \texttt{t_create} forks a thread for the planner (currently we use WARPLAN [War74]) to run concurrently to the perception-reaction-cycle. The perception-reaction-cycle pops events from the event queue by calling \texttt{pvm_recv}. Subsequently it generates a reaction in response to the event. The reaction comprises besides assimilating epistemic effects, performing actions which are available as meta predicates and sending messages which are translated to \texttt{pvm_send}. The knowledge base is an extended logic program [AP96] being easily implementable in Prolog. Extended logic programming is very expressive since it offers two kinds of negation and integrity constraints.

12
6 Experiments

Applications of purely reactive vivid agents include distributed diagnosis [SiAMP96, SW96, FdAMNS96]. However, to reflect the deliberative and pro-active capacities we illustrate the above implementation and concepts by the following example: An agent a sends a task to agent b and monitors b’s progress until b finished its task. In this case b is supposed to plan and execute moving actions in the blocks world as depicted in figure 6.

The agents a and b are specified as follows:

- a sends b on receipt of an initial message by the creator process, the command to plan to achieve that block a is on top of b which is on top of c.

- On receipt of the initial message by the creator, a also sends a request for information about the b’s progress.

- a records the progress of b and continues monitoring b until b has accomplished the whole task.

- If agent b receives a command from a, it performs it. In particular, it serves a’s requests for information on the state of affairs.
Figure 6: Agent a gives agent b a planning task and observes b’s progress.

Formally, we can translate the above description into the following action and reaction rules and knowledge base.

Master a

Reaction Rules

sendMsg(do(plan,goal(on(a, b) & on(b, c))), b) ← recvMsg(tell(start), creator).
sendMsg(request(on), b) ← recvMsg(tell(start), creator).
accomplished_part1 ←
    recvMsg(reply(on(b, c), T), b), not accomplished_part1.
accomplished_part2 ←
    recvMsg(reply(on(a, b), T), b), not accomplished_part2.
sendMsg(request(on), b) ←
    recvMsg(reply(on), b), not accomplished.

Action Rules

Knowledge Base

accomplished ← accomplished_part1, accomplished_part2.
Slave b

Reaction Rules
\[ \text{do}(A) \leftarrow \text{recvMsg}(\text{do}(A), a). \]
\[ \text{sendMessage}(\text{reply}(\text{on}(A, B), T), a) \leftarrow \text{recvMsg}(\text{request}(\text{on}), a), \text{on}(A, B), \text{time}(T). \]
\[ \text{sendMessage}(\text{reply}(\text{on}), a) \leftarrow \text{recvMsg}(\text{request}(\text{on}), a). \]

Action Rules
\[ \text{move}(U, V, \text{floor}) : \neg \text{on}(U, V, \text{floor}), \text{clear}(V) \leftarrow \text{on}(U, V), V \neq \text{floor}, \text{clear}(U). \]
\[ \text{move}(U, V, W) : \neg \text{on}(U, V), \neg \text{clear}(W), \text{on}(U, W), \text{clear}(V) \leftarrow W \neq \text{floor}, \text{clear}(W), \text{on}(U, V), U \neq \text{W}, \text{clear}(U). \]

Knowledge Base
\[ \text{on}(a, \text{floor}), \text{on}(b, \text{floor}), \text{on}(c, a), \text{clear}(b), \text{clear}(c). \]

If executed on the PVM platform the above specification can lead to a trace as depicted in figure 7.

<table>
<thead>
<tr>
<th>Step</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b ← a</td>
<td>\text{do(plan,goal}(\text{on}(a,b)&amp;\text{on}(b,c)))</td>
<td>a ← b \text{reply}(\text{on}(b,\text{floor}),9109815)</td>
</tr>
<tr>
<td>b ← a</td>
<td>\text{did plan,goal}(\text{on}(a,b)&amp;\text{on}(b,c))</td>
<td>a ← b \text{reply}(\text{on}(c,\text{floor}),9109815)</td>
</tr>
<tr>
<td>b ← a</td>
<td>\text{request(on)}</td>
<td>b \text{executes move}(b,\text{floor},c)</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(a,\text{floor}),9109812)}</td>
<td>a ← b \text{reply(on(a,\text{floor}),9109812)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(b,\text{floor}),9109812)}</td>
<td>b \text{assimilates neg on}(b,\text{floor})</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(c,a),9109812)}</td>
<td>b \text{assimilates neg clear}(c)</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on)}</td>
<td>b \text{assimilates on}(b,c)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>b \text{assimilates clear(floor)}</td>
</tr>
<tr>
<td>b ← a</td>
<td>\text{request(on)}</td>
<td>b ← a \text{request(on)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(a,\text{floor}),9109815)}</td>
<td>a ← b \text{reply(on(a,\text{floor}),9109815)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(b,\text{floor}),9109815)}</td>
<td>a ← b \text{reply(on(b,\text{floor}),9109815)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(c,a),9109815)}</td>
<td>a ← b \text{reply(on(b,c),9109816)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on)}</td>
<td>a \text{assimilates accomplished part1}</td>
</tr>
<tr>
<td>b</td>
<td>\text{executes start}</td>
<td>b \text{executes move}(a,\text{floor},b)</td>
</tr>
<tr>
<td>b ← a</td>
<td>\text{request(on)}</td>
<td>b \text{assimilates neg on}(a,\text{floor})</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(a,\text{floor}),9109815)}</td>
<td>a ← b \text{reply(on(a,\text{floor}),9109815)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(b,\text{floor}),9109815)}</td>
<td>b \text{assimilates on}(a,b)</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(c,a),9109815)}</td>
<td>b \text{assimilates clear(floor)}</td>
</tr>
<tr>
<td>b</td>
<td>\text{executes move}(c,a,\text{floor})</td>
<td>a ← b \text{reply(on)}</td>
</tr>
<tr>
<td>b</td>
<td>\text{assimilates neg on}(c,a)</td>
<td>b ← a \text{request(on)}</td>
</tr>
<tr>
<td>b</td>
<td>\text{assimilates on}(c,\text{floor})</td>
<td>a ← b \text{reply(on(c,\text{floor}),9109816)}</td>
</tr>
<tr>
<td>b</td>
<td>\text{assimilates clear}(a)</td>
<td>a ← b \text{reply(on(b,c),9109816)}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on)}</td>
<td>a ← b \text{reply(on(a,b),9109816)}</td>
</tr>
<tr>
<td>b ← a</td>
<td>\text{request(on)}</td>
<td>a ← b \text{assimilates accomplished part2}</td>
</tr>
<tr>
<td>a ← b</td>
<td>\text{reply(on(a,\text{floor}),9109815)}</td>
<td>a ← b \text{reply(on)}</td>
</tr>
</tbody>
</table>

Figure 7: A trace. An entry \( x \leftarrow y \) means that \( x \) received from \( y \) the message \( z \). For the sake of brevity we left out send messages (note that communication is asynchronous) and the creator’s start and halt messages.
7 Related Work

7.1 IC-Prolog II

IC-Prolog II is a full Prolog system which encompasses several features supporting parallel and distributed logic Programming. Here we compare its approach to support distributed agent systems [Chu93] with PVM-Prolog.

Within a Prolog process IC-Prolog II basically offers the same concepts as PVM-Prolog. It allows asynchronous execution of multiple Prolog threads, each resolving independent goals. Those threads communicate through pipes the same way as PVM-Prolog threads communicate through term queues.

In a networking environment, IC-Prolog II (as most widely used Prolog systems) supports communication through sockets, offering a predicate library interface, whereas PVM-Prolog offers a full-fledged PVM interface library.

Sockets are good for network communication and they are widely accepted. However, their use involves dealing with low-level details such as finding the communication partner, binding the socket and handling different data formats. PVM supports a higher level communication model, where name resolution and port allocation are hidden from the programmer. It also supports multi-casting, group communication, and synchronization mechanisms. PVM also covers many multi-processor architectures where sockets are not available.

In contrast to IC-Prolog II, PVM supports heterogeneous hardware configurations with different data formats, which has two advantages: First, the internal data format allows a more efficient representation than the textual one and it makes communication among processes created by different languages possible. Second, PVM's host configuration can be changed dynamically, an issue which is not handled in IC-Prolog II.

Another difference concerns IC-Prolog's indirect naming of mailboxes and PVM-Prolog's direct naming of the communication partner. IC-Prolog's mailboxes also support multiple readers. In PVM the group communication mechanisms can be used for a similar purpose.

7.2 April

April [MC95b] is a language which supports a model of execution similar to CSP [Hoa85] using pattern matching to check for incoming messages. It also includes real-time support, and higher-order features such as lambda abstractions, and a macro pre-processor.

The communication model is point to point, with direct naming of the communicating partners. It also supports process creation and global naming through a structured name service for long range communication, which mirrors the traditional DNS.

Though April includes ideas from logic programming such as pattern matching of messages, it does not focus on declarative and executable high-level specifications. It is rather a mix of several paradigms. Concerning April's level of
abstraction it ranges between PVM-Prolog which is a pure paradigm extended by message passing and a high-level multi-agent language such as vivid agents.

### 7.3 ICE

ICE [Am95, AB96] defines a model to support distributed AI applications over hybrid languages and heterogeneous, distributed platforms. It is available from Prolog, Lisp, C, C++ and Tcl/Tk and is similar to PVM-Prolog, implemented on top of PVM.

Its communication model is point to point through named channels but it also supports broadcasting of messages to all components. Components may be created dynamically but their communication topology must be defined in a configuration file. Configuration of components and their communication channels is done through a configuration file by a special process, the license server.

PVM-Prolog, as ICE, also supports communication with tasks written in different languages. ICE’s communication model is defined at a higher level than PVM-Prolog’s. However it is necessary to specify the topology of the communication before running the system - this is a clear limitation in contrast with PVM dynamic configuration, communication and process control features.

### 7.4 Other Distributed Logic Programming Languages

The first distributed logic programming language to appear in the literature is Delta-Prolog[LM84, CMP92]. Delta Prolog supports a declarative distributed logic programming model which requires distributed backtracking mechanisms which are difficult to implement. Other distributed logic programming systems such as PMS-Prolog[WJH92] and Shared-Prolog[BC91] elude the need for distributed backtracking either by using message passing extra-logical predicates in the former or preventing it in the language syntax in the latter. Shared-Prolog relies on the LINDA shared space communication model while PMS-Prolog and PVM-Prolog use distributed space communication models. Neither Shared-Prolog or PMS-Prolog support multi-threading of a Prolog process. Delta-Prolog and PMS-Prolog have been implemented in transputer multi-computers Delta-Prolog has also been implemented in UNIX systems connected through local area networks.

### 8 Conclusion

To conclude let us compare which of the requirements mentioned in the introduction are met in the current implementation of vivid agents and how the underlying PVM-Prolog contributes.

**Reactive and Pro-active Behaviour** The agents’ reactive and pro-active behaviour can be implemented by using PVM-Prolog’s thread concept to run a
perception-reaction-cycle concurrently to a planning facility. Such light-weight concurrency has the advantage of fast communication between strongly connected parts of a single agent.

**Formal Semantics** The formal semantics of the CAP architecture helps understanding and allows program verification. The formal semantics is independent of PVM-Prolog as implementation language, but it was influenced by the concepts provided by PVM-Prolog and therefore easily implementable in PVM-Prolog.

**Executable Specifications** The idea of Prolog executing specifications is lifted by the vivid agent concept and its implementation in PVM-Prolog to a higher and more declarative level of multi-agent specifications.

**Hardware Independence** PVM is hardware independent and available for a range of architectures including PCs, Workstations and Multi-computer platforms. PVM's and PVM-Prolog's hardware independence also makes vivid agents more flexible.

**Openness** PVM is best suited for local area networks. A more appropriate global network interface is not yet implemented. Currently we are experimenting with Email. We included a primitive to send mails into PVM-Prolog and translate incoming emails addressed to the MAS to PVM messages.

**Heterogeneity** PVM has been widely used up to now, so many tools and different language interfaces are available. The PVM distribution supports C and Fortran; interfaces for C++, Ada, Lisp, Perl, Tcl, and Prolog have subsequently been developed. First results show that heterogeneous agents can run together in a net but that the MAS configuration has to be public to ensure that agents can use the proper formats to communicate. A possible solution is a public yellow pages service, which might be implemented itself in PVM-Prolog.

**Acknowledgements**
We would like to thank Wolfgang Nejdl and Luís Moniz Pereira for their support. Financially, the work was supported by the German BMBF, the Portuguese JNICT, and the PRAXIS PROLOPPE project.

**References**


19


