CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

Simulating Actions of Autonomous Agents

by
Claude Le Pape

TECHNICAL REPORT
Number 38

October, 1990

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

FOREWORD

The work presented in the following report has been performed in the context of two projects aimed at controlling the actions of multiple agents in the same environment.

- **Indoor Automation With Many Mobile Robots.** The goal of this project is to control the operations of many mobile robots (several dozens) in an indoor environment (an office environment, a shop-floor, an airport, an hotel) in order to automate a variety of tasks. Typical tasks include transportation of objects (beverages, books, mail), operation of machines (copiers, vending machines), cleaning and maintenance. Different tasks may require different physical capabilities. Nevertheless, most of the tasks essentially involve mobility and transportation of relatively small objects.
- **Planning, Scheduling and Monitoring Actions of Multiple Agents on a Construction Site.** This project concerns the integration of various short-term planning, scheduling and execution monitoring techniques for multiple agents (robots and humans) working on a construction site. The case of a construction site introduces additional difficulties. For example, the geometry of a construction site continually changes. More sophisticated planning techniques (integrating temporal and geometrical reasoning) are consequently needed.

The current planning, scheduling and execution system integrates a collection of software components: a **task planning system** to derive plans made of "high-level" actions (such as "go to position P" or "get object O") from a description of the tasks to be performed; a **task allocation system** to order and allocate tasks or actions to agents; a **motion planning system** to convert high-level actions into motion commands; and an **execution system** to monitor execution and react to unexpected events. These components are implemented in COMMON-LISP on a DEC 3100 workstation. The overall system is tested with the help of a simulation system designed to simulate actions of autonomous agents.

This report presents the simulation system. A second report (in preparation) will discuss the use of constraint-based planning and scheduling techniques in the context of these two projects.

Simulating Actions of Autonomous Agents

Claude Le Pape

Robotics Laboratory
Department of Computer Science
Stanford University
Stanford CA 94305

Abstract: We present a simulation system specially designed to simulate actions of autonomous agents in partially unpredictable environments. This system results from considering the problem of developing a simulation system for multiple agents without making any restrictive assumptions about the nature of the environment and about the cognitive behavior of agents. The use of the system allows to make experiments with various robot softwares (e.g. with combinations of task and motion planning algorithms), to determine to what extent an agent architecture allows a robot to efficiently react to unexpected events, to compare multi-agent cooperation frameworks and to test the transient and asymptotic behaviors of agents having imperfect knowledge about their environment. Currently, the most advanced application concerns the use of centralized and distributed multi-robot planning and scheduling techniques in an office environment [36].

Keywords: Simulation, reaction to unexpected events, cognitive action, planning, scheduling, distributed artificial intelligence, blackboard system.

Acknowledgements: This research has been funded by the Center for Integrated Facility Engineering (CIFE), the Center for Integrated Systems (CIS) and Digital Equipment Corporation (DEC). A short version of this paper has been published in 1990 in the Proceedings of the AAAI Workshop on Artificial Intelligence and Simulation [37].

Contents

"I can call spirits from the vasty deep. Why so can I, or so can any man; but will they come when you do call for them ?"

William Shakespeare (King Henry IV).

1 Motivation

The work presented in this paper has been performed in the context of two CIFE projects aimed at controlling the operations of many **autonomous** robots (several dozens) in partially **unpredictable** environments. By autonomous, we mean that robots are able to convert a high-level description of the tasks to be performed into robot actions (typically, motions and sensing) with no human assistance. Thus, autonomous robots allow their users to specify **what** has to be done rather than **how** to do it. By unpredictable, we mean that robots cannot completely determine in advance which events are going to occur and when. Robots must be able to react on-line to a variety of incidents: unexpected obstacles, broken tools, unexpected delays, uncoming answers, discharged robot batteries and robot breakdowns.

Many difficult problems must be solved to operate dozens of autonomous robots in the same environment: implementation of non-conflicting sensor systems, man-robot and robot-robot communication systems and protocols, contingency-tolerant motion control, multi-robot motion planning, multi-robot task planning and scheduling. In the meantime, using dozens of robots to make extensive experiments with research softwares is simply not cost-effective: it implies both buying and maintaining many robots and dedicating space to experiments for a long period of time. On the other hand, the usual solution which consists in running only a few experiments (designed to be particularly interesting) is very tedious given the desired number of robots. Indeed, it implies the explicit generation of interesting **scenarios**. A scenario is a sequence of incidents associated with a sequence of robot actions (see [21] for a good example). Its design consists in deciding which robot actions go right and which go wrong, e.g. how a robot deviates from its planned trajectory, which breakdowns occur, which pieces of information are not obtained from sensors. When many robots act in parallel, a realistic scenario contains many incidents. Both its design and execution take a lot of time.

Things are even more complex when one wants to compare softwares. For example, researchers designing a new reactive planner generally want to test the new planner against other planners. To do this, one may just consider running the same scenario for each planner. But, in most cases, this happens to be either scientifically incorrect or practically impossible. Indeed, since different planners do not provide the same plans in the same situation, incidents disturbing plan execution cannot be designed to have equivalent effects from one planner to the other. Therefore, many experiments must be made to allow experimental comparisons between planners. Since the definition and the monitoring of n execution scenarios is not a very interesting task, we prefer to have such scenarios randomly generated with the help of a simulation system.

Consequently, we developed a simulation system to simulate the actions of many agents in unpredictable environments.[1] The use of the system allows to make experiments with various robot softwares (e.g. with combinations of task and motion planning algorithms), to determine to what extent an agent architecture allows a robot to efficiently react to unexpected events, to compare multi-agent cooperation frameworks and to test the behavior of agents having imperfect knowledge about their environment. More generally, it allows to test reactive system designs in a great variety of situations, and not only (as is often the case) in a few well-defined situations. It complements the actual operation of a few robots in a controllable laboratory setup.

In section 2, we discuss the desired properties of a simulation system designed to simulate actions of autonomous agents. In section 3, we provide an overview of our system. In sections 4, 5 and 6, we discuss the representation of the environment, the description of actions occurring in this environment and the simulation of these actions. Current and expected applications are presented in section 7.

[1]Throughout the paper, we will use the expression "simulation system" rather than "simulator" to stress the fact that the system is not designed for a unique application.

2 Desired Properties

An obvious requirement of a simulation system is that it must be general. We want the system to be applicable in many cases and (applicability is not enough[2]) appropriate for many applications. This means the system must be easy to use and allow the user to describe actions with the precision that is needed for each application. In addition, the simulation system must provide unbiased results and remain efficient, even though there are unavoidable tradeoffs between efficiency, naturalness and precision. We exhibit four important issues to consider in order to meet these requirements in a simulation system designed to simulate actions of autonomous agents: simulation of cognitive actions (section 2.1), description and simulation of actions at various levels of abstraction (section 2.2), simulation of events occurring without involving agents (section 2.3) and simulation of deviations and unexpected events (section 2.4).

2.1 Simulation of Cognitive Actions

One of the most important difficulties identified by researchers in the domain of reactive planning and scheduling is that the environment in which a planning agent evolves changes as the agent evolves. For example, the environment may change while the planning agent constructs a plan, thereby making the plan inapplicable. For small applications, this phenomenon is negligible. However, a short analysis shows that its rate of occurrence grows much quicker than the rate of relevant events in the environment. In [10], Collinot and Le Pape report that the average number of events occurring in a factory while the SONIA scheduling system modifies factory schedules multiplies by ten when the total rate of events doubles (figure 1). Considering the design of a reactive agent, this means that the planning and plan execution activities must not prevent the agent from integrating (and reacting to) information about the occurrence of unscheduled, asynchronous events (see [23] [32] [34] [43] [45]). From the practical viewpoint of simulating the actions of dozens of agents, this means that the design of the simulation system must not preclude cases in which the performance of a cognitive action overlaps the occurrence of an event relevant to this cognitive action.

If we are concerned with the operations of n computing processors, an obvious approach to correctly simulate such cases consists in using a total of $(n + 1)$ computing units, n to effectively accomplish the computing operations and 1 to simulate the evolution of the environment. Such a solution is simple for small values of n. But it requires the use of $(n + 1)$ computers and the management of communications among them. It becomes impracticable when n increases.

[2]See Chandrasekaran's comments on general architectures in [27]: "It does not take much for an architecture to be Turing universal, but Turing universality does not guarantee that the architecture is appropriate in the sense that it supports in a natural way what we want to say and how we want to say it ... The danger of the pure task-specific view is a proliferation of architectures that are difficult to integrate, while the pitfall of the general architecture proposals is to mistake mere Turing power for the kind of generality needed."

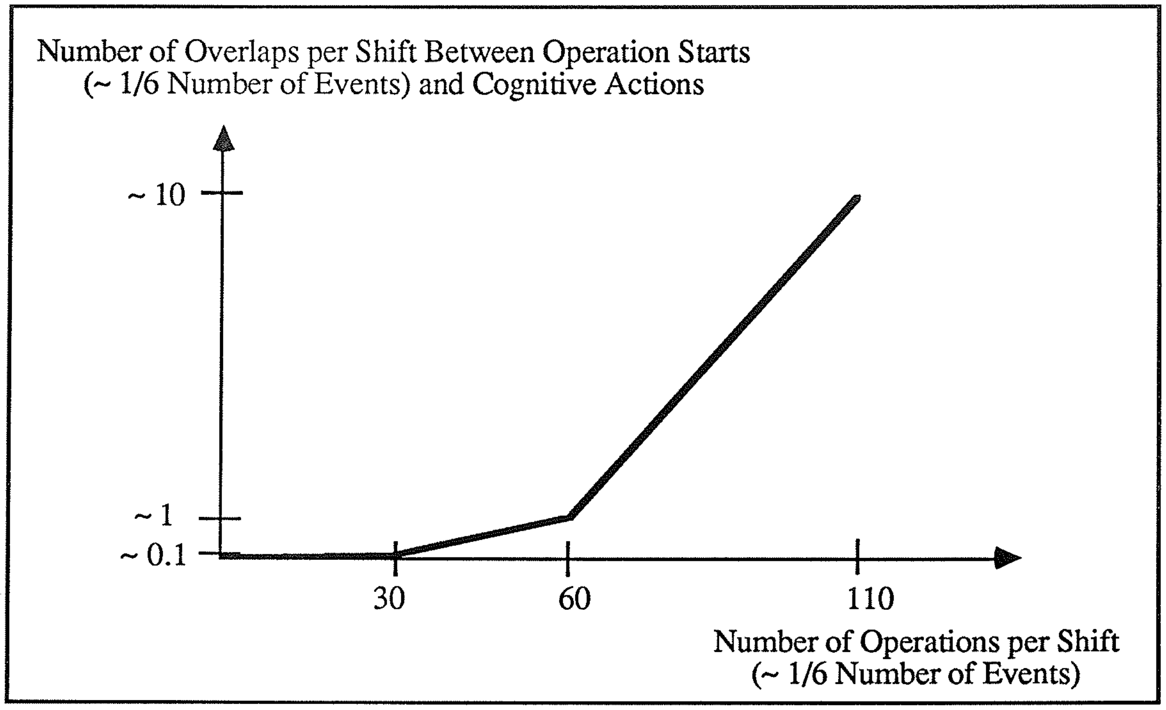


Figure 1: Overlaps Between Events and Cognitive Actions in SONIA

Another approach consists in coordinating the execution of $(n + 1)$ dependent processes on the same computer (or on a limited number of computers). In most cases, knowledge about the interactions between an agent and its environment allows to make the coordination simple. For example, the coordination technique described in [10] for the SONIA scheduling system rests upon a simple fact: an event does not need to be simulated before any agent is ready to take it into account. Nevertheless, the use of such a technique requires a prior analysis of the interactions between each agent and its environment (including other agents). The generalization of such a technique into a general simulation system requires the automation of this analysis, which is a very complex task.

The approach we advocate consists in simulating cognitive actions. This means there is a single simulation process. **Both the computations performed by the n agents and the evolution of the environment are simulated.** The interest of this approach resides in the absence of distinction between cognitive and physical actions. However, this absence of distinction implies that one must bridge the gap between the execution of a computer program and the simulation of this execution. Then, the difficulty is to implement a simulation system (a) without making too restrictive assumptions about the cognitive behavior of agents, (b) without making it too difficult for the system user to transform an algorithm into a description enabling the simulation of the algorithm execution and (c) without making the simulation of an algorithm execution too costly compared to the execution itself. We will see in this paper how our implementation satisfies these requirements for a variety of cognitive processes.

2.2 Alternatives in Action Descriptions

A general simulation system must allow the description of very different types of actions, events and processes. For example, there are important differences between experimentations concerning task and motion planning algorithms.

- Task planning consists of deriving programs of high-level actions (such as "go to position P" or "take object O") from a description of goals to be achieved. Simulating discrete changes occurring as the result of robot actions is often sufficient to test multi-robot task planning systems.

- The motion planning and control problem consists of (a) converting high-level actions into motion commands and (b) monitoring the execution of the motion commands. Simulating continuous robot motions is necessary to test reactive motion planning systems. In some cases, one may "discretize" the simulation of motions, computing at each step the motion of each robot from time t_i to time $t_{i+1} = t_i + \Delta$. The problem with such an approach is to choose Δ small enough for the simulation to be correct and large enough for the simulation to run quickly. In some cases, the user of the simulation system will rather provide functions to determine the exact points in time at which interesting events (such as the crossing of two robots) occur, thereby representing the real nature of continuous processes.

More generally, simulating the details of a process is either necessary or superfluous depending on the system user's needs. Therefore, the system must allow the user to choose an appropriate level of abstraction for each physical and cognitive action.

Relating to this choice, one of the most difficult issue is to represent the effects of concurrent actions. Indeed, these effects may, at some levels of abstraction, bear little resemblance with the effects of any one action in isolation. For instance, a robot carrying an object in a corridor while another robot is painting the walls of the corridor will not arrive to its destination. It will change color and stop functioning (in addition, the walls will not be correctly painted). To take such an interaction into account, one may either describe processes in details or provide the simulation system with knowledge enabling the detection of interactions and the computation of effects of interactions. We believe that in many cases this knowledge can be expressed easily and in a general fashion. For example, a collision between two objects is an interaction which occurs only when at least one object moves until it makes contact with another object. Therefore, it is not necessary to simulate the motions of n objects millisecond by millisecond when we can compute the exact dates at which objects collide as well as the consequences of the collisions. Furthermore, when exact computations are too complex, one can still provide functions to compute intervals of time during which interactions occur and detail simulations over these intervals.

In many cases, interruptions of actions can also be simulated without detailing the simulation of interruptible actions. For instance, the position of a robot which moves in a corridor and interrupts its motion at time t can be computed without making a millisecond by millisecond simulation of the robot motion. Knowing the new position of the robot, a subsequent resumption of the motion can also be simulated without decomposing the motion into minuscule chunks of motions. Consequently, we must allow the user of the simulation system to provide an efficient representation of the effects of interrupted actions.[3]

2.3 Simulation of Processes Without Agents

In the previous sections, the discussion was focused on the representation of actions explicitly designed and performed by agents operating in the simulated environment. In reality, we often consider that many events and processes occur without actually involving agents. For instance, when a bucket is under an open tap, it fills up without having any agent performing the action of filling it up. Similarly, we consider that robots break down without determining any agent breaking them down. Such events are either **predictable** or **unpredictable**. In this section, we will discuss the simulation of predictable events. By predictable, we mean events which we can expect to occur given a set of rules governing the normal evolution of the environment. We will discuss the simulation of unpredictable events — the occurrence of which is not governed by a known deterministic set of rules — as part of section 2.4.

The simulation system must provide a way of defining rules governing the occurrence of predictable events. From the point of view of designing the simulation system, the main problem is to realize some conceptual consistency between the simulation of actions and the simulation of predictable events. Not only do we agree with Brooks [5] that conceptual consistency is "the most important consideration in system design", we also notice that in this particular case the importance of conceptual consistency is reinforced by two noticeable facts:

- The representation of rules governing predictable events raises issues similar to those discussed in section 2.2. For instance, the user of the simulation system may either want to detail the simulation of the bucket filling process or provide functions to determine when water begins to overflow the bucket.
- Interactions exist between actions and processes not driven by agents. A simple example (already chosen in [24]) is when an agent takes some amount of time t to progressively open (or shut) a valve. In this case, the content of the bucket at the end of the opening (or shutting) action may be computed with the help of the equation (*final-content = initial-content + initial-flow-rate*t + turn-rate*t^2/2*).

[3]Since the interruption and the resumption of an action are also actions, we could consider interruptions as a particular case of interactions. However, the significance of this particular case incites to provide more practical structures to deal with it.

This suggests to design the simulation system so that the user provides similar descriptions for actions (involving agents) and for processes occurring without involving agents. An obvious solution is to get rid of the action concept and consider agents as mere moving and computing devices. However, the simulation system is intended to make experiments with cognitive architectures and components of computing devices explicitly considered as agents. Consequently, a better solution is to conserve the action concept and require action-like descriptions for processes occurring without involving agents. What we will do is create fictive agents and do as if they were executing those processes not executed by actual agents.

2.4 Simulation of Deviations and Unexpected Events

As mentioned in section 1, we want to simulate actions of agents able to react to a variety of incidents. With respect to simulation difficulties, we distinguish three categories of incidents.

Unexpected Facts/Events/Actions

This is the easiest case in which an agent plans its own actions without considering (or without knowing) the current state of the environment and the current plans of other agents. Typical examples are the discovery of an unexpected obstacle and the unforeseen competition of two agents for the same resource. There is nothing special to do to simulate the occurrence of such incidents. They are direct consequences of the ways agents are designed to operate.

Deviations

This category gathers differences between the prediction of an agent with respect to some actions and the actual execution of these actions: the performance of an action does not take the exact amount of time the agent was expecting to devote to it; intended effects do not follow from an action; etc. One needs to provide information about the possible occurrence of these incidents in action descriptions. For example, instead of providing a deterministic function to compute the duration of an action, the user of the system may describe the statistical distribution of this duration.

Unpredictable Events

Unpredictable events are not governed by known deterministic sets of rules and not associated with the performance of particular actions. A typical example is when a robot component starts malfunctioning or not functioning at all. In most cases, the user of the simulation system will want to simulate a few **most likely** sensing, motion and cognitive problems — and the combined effects of these problems — without developing deep models of the overall process going on within a disturbed agent. Following previous arguments (in section 2.3 and above), the simulation system must allow action-like descriptions of unpredictable events, the occurrence and the characteristics of which are controlled by statistical distributions. Instead of providing deterministic rules governing the occurrence of predictable events, the task of the user will be (for example) to specify the frequency of some machine breakdowns with respect to machine utilization, as well as the distribution of breakdown durations.

3 An Overview of the Simulation System

Basically, the simulation system must provide ways to (a) represent possible states of an environment (including agents), (b) describe actions and processes occurring in this environment and (c) simulate the occurrence of these actions and processes. More precisely, the essential expectation of system users is a *simulation* function allowing to simulate the evolution of an environment from a description of this environment, e.g. a function of the following type:

- Function Name: *simulation*
- Arguments: (*environment*)
- Optional Arguments: (*duration stop-test-function*)
- Default Values: ((*duration* $= \infty$) (*stop-test-function* $=$ *false*))
- Action: simulate the evolution of the environment either for the given duration or until (*stop-test-function environment*) returns true.
- Remark: when *duration* and *stop-test-function* take default values, the simulation cannot end unless the environment gets petrified in a particular state.

On the other hand, although the user expects to have to describe the environment in a particular formalism (so that the *simulation* function can be applied), the system should in this matter provide the greatest flexibility. Concerning the representation of possible states of the environment, we will see that, in fact, very few pieces of information need to be expressed in the simulation formalism. We consider it quite important that the system allows the user to use any representation scheme to represent other pieces of information. The main goal of a simulation system is not to provide a rigid universal model to represent the world. It is to provide a convenient *simulation* function.

The representation of the environment is discussed in section 4. The user of the system provides the initial date, the initial set of agents and the set of actions being performed by each agent in the initial state. The execution of the *simulation* function results in updating this information. In section 5, we discuss the representation of actions and interactions in terms of **change functions**. These functions determine **dates of interest** for each action (and interaction) and describe how each action (and interaction) affects the environment between two dates of interest. The *simulation* function is described in section 6. It consists of a three-step simulation loop. For each cycle, the system (1) computes a list of ongoing actions and forthcoming interactions, (2) determines actions (or interactions) for which the next date of interest is minimal and (3) alters the representation of the environment in accordance with the change functions.

4 Representation of a State of the World

Very few pieces of information about the state of the world have to be expressed in the simulation formalism. Basically, the system needs to know (a) the date, (b) the set of existing agents and (c) the set of actions being performed by each agent in the considered state. This includes the set of natural and unexpectable processes executed by fictive agents (cf. sections 2.3 and 2.4).

Date

The simulation system allows the user to choose any totally ordered set S as a model of time. Dates are members of S and the user provides a definition of the ordering relation in the form of two functions *time=* and *time≥*. To manipulate the concept of duration (which is not compulsory), the user provides an additional function *time+* allowing to add dates and durations. This is similar in spirit to the kind of modelling flexibility discussed in [35] and allows us to disregard many irrelevant debates such as those surrounding the distinction between discrete and continuous models of time. To present examples, we will in this paper choose the set of rationals (with its usual ordering relation) as a date space (neither discrete, nor continuous) and the set of positive rationals (with the usual addition) as a duration space. Furthermore, we will assign the date 0 to the initial state.

Agents

The system must be provided with a description of existing agents. This includes actual agents (performing actions) and fictive agents governing the occurrence of processes which do not involve actual agents. Each agent is assigned a name and a set of ongoing actions. During the course of simulation, agents will complete actions, generate and initiate new actions, cancel actions, suspend actions, and resume actions. The performance of some actions can result in the destruction of agents. It can also result in the creation of new agents to which new names must be given. Except *name* and *actions*, all the characteristics of agents will not be directly accessed by the *simulation* function. They are application-dependent and the user of the system can use any formalism to represent them. Similarly, the user can use any formalism to represent every thing which is in the environment without being an agent.

Actions

Within the description of an agent, the description of an action consists of five attributes.

- Name: the name of the action.
- Agent: the corresponding agent.
- Status: the status of the action. The major values of this attribute are *pending*, *cancelled*, *inprocess*, *completed* and *suspended.* Figure 2 presents the sensible transitions between two of these values. In addition, it is convenient to introduce two transitory values *starting* and *restarting.* These transitory values correspond to the transition from *pending* to *inprocess* and to the transition from *suspended* to *inprocess.* The introduction of these values allows to distinguish the decision (transmitted to the corresponding effector) of an agent to start or restart an action and the simulation of the actual beginning or resumption of this action (figure 3).
- Type: the type of the action.
- Parameters: the values of parameters associated with the action.

To each action type, the user of the system must associate a change function allowing to determine the dates of interest of each action and the changes occurring between two dates of interest. *Completed* and *cancelled* actions will not have any date of interest in the future. They can be discarded and gathered in the history of the environment. *Starting* actions are about to start, *restarting* actions are about to restart and *inprocess* actions are executing. The first date of interest of these actions, in process or scheduled to start in the initial state, is the date assigned to the initial state. *Pending* and *suspended* actions have no initial date of interest. They will get a first date of interest when the agent decides to start or restart them and consequently updates their status (respectively to *starting* and *restarting*). Except *name*, *agent*, *type*, *parameters* and *status*, all the characteristics of actions will not be directly accessed by the *simulation* function. They are application-dependent and the user of the system can use any formalism to represent them.

Figures 4 and 5 provide examples of initial state descriptions. Figure 4 is an example in which the user extends the object-oriented representation of the simulation system. Figure 5 is an example in which the user prefers to use propositional formulas. Only necessary information is provided in the object-oriented representation of the simulation system. In both cases, the execution of the simulation function will result in updating the state description with respect to change functions.

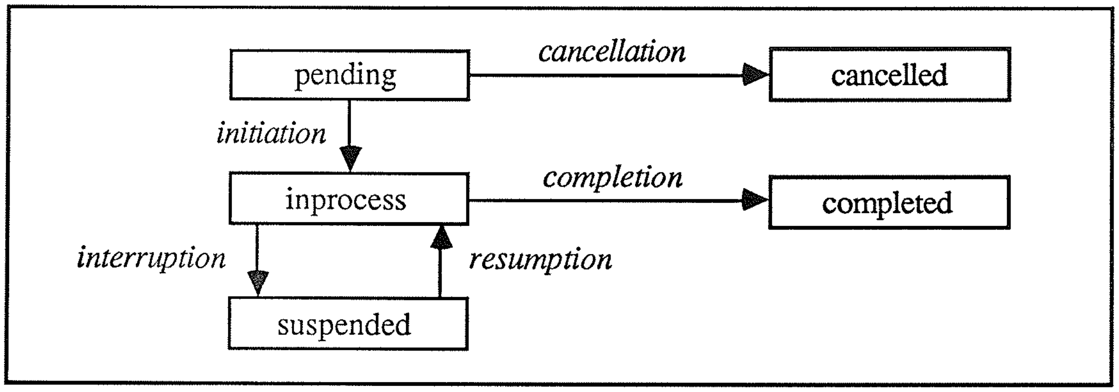


Figure 2: Transitions Between Major Action Statuses

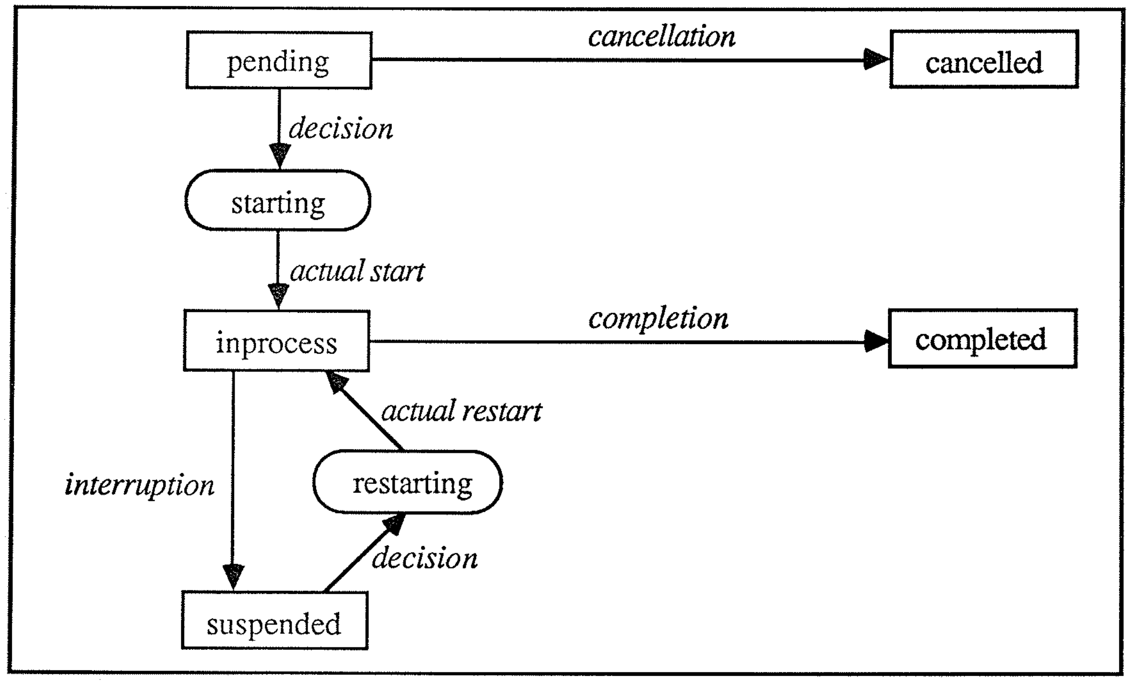


Figure 3: Transitions Between Action Statuses

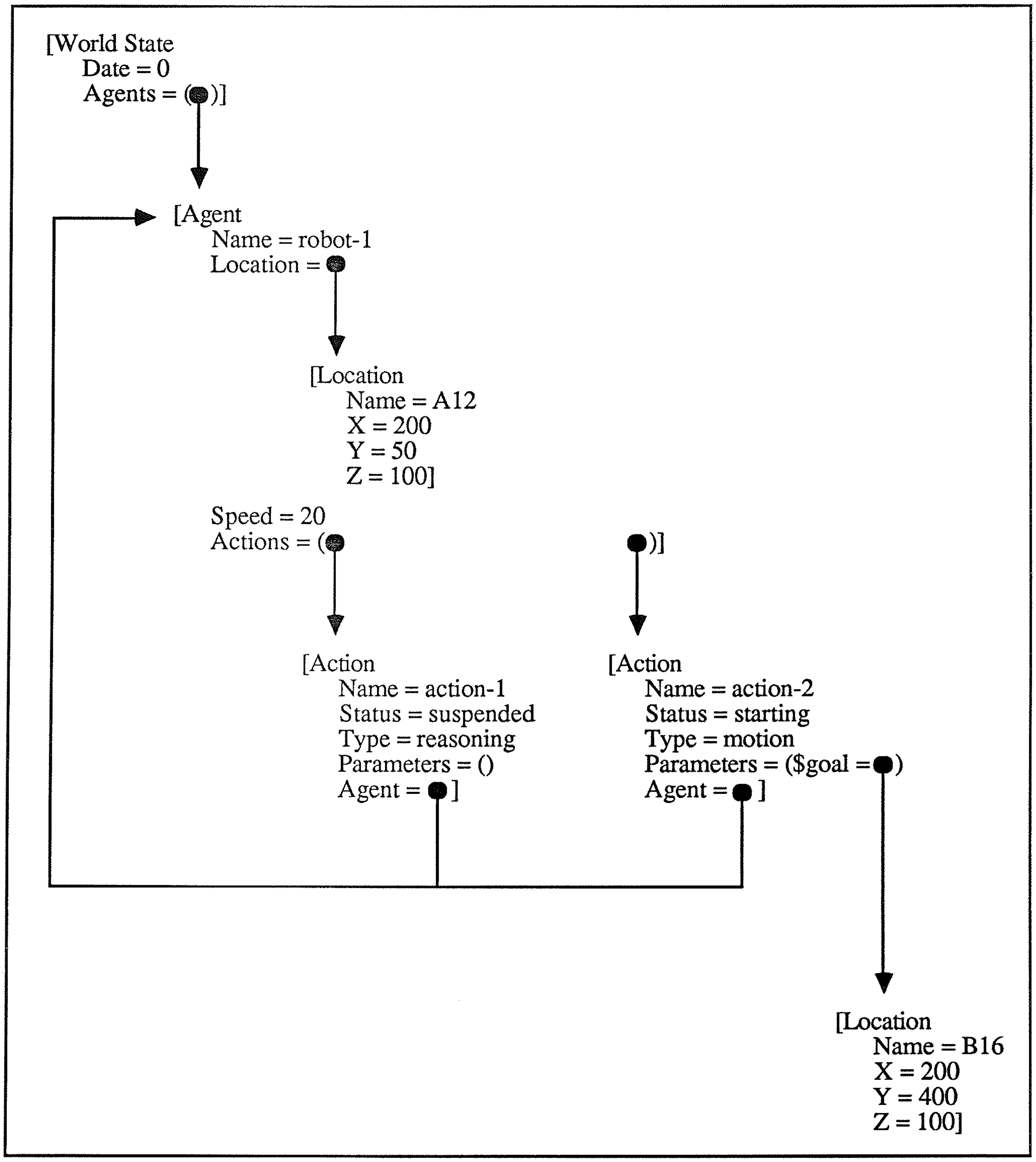


Figure 4: An Object-Oriented State Description

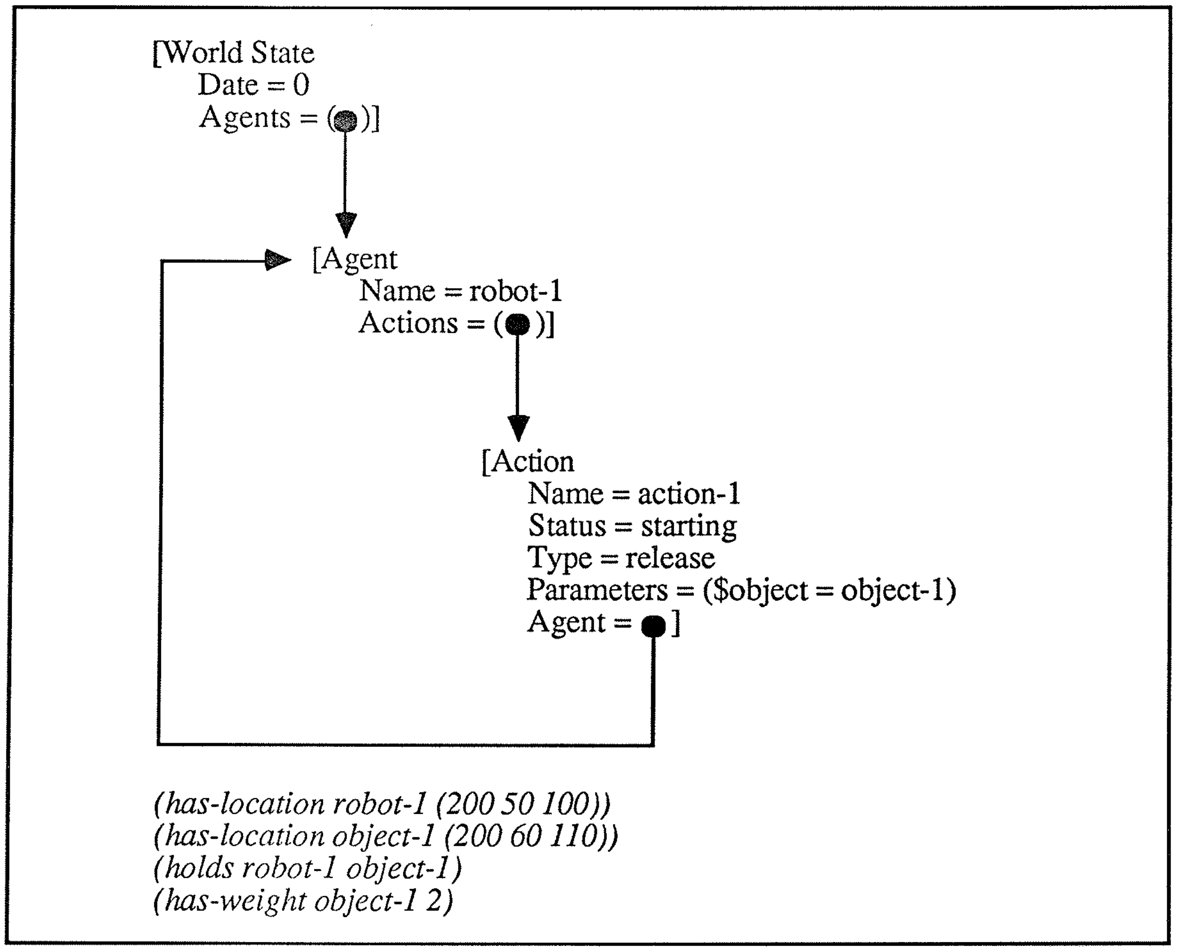


Figure 5: A State Description With Propositional Formulas

5 Change Functions for Actions and Interactions

5.1 General Framework

Since actions interact, a correct simulation of both physical and cognitive actions requires an answer to a variety of questions, such as:

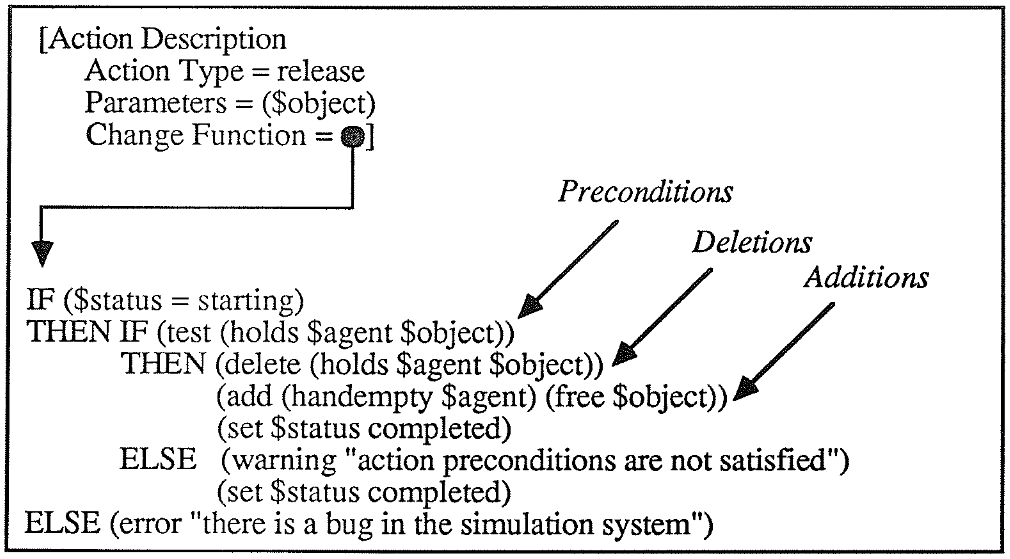
- Are physical and cognitive actions performed in parallel ?
- Which are the smallest non-interruptible physical and cognitive actions of an agent ?
- When does an agent incorporate sensory information into its line of reasoning ?

As a matter of fact, reasoning about simulating the actions of autonomous agents in unpredictable environments gives rise to many questions concerning the types of behaviors these agents may have. A discussion of these questions is outside the scope of this paper. We suppose that the user of the simulation system can provide an answer to these questions. In this section, we discuss the subsequent description of actions and interactions.

When the simulation begins, the system associates a first date of interest (the current date) to each *starting*, *restarting* or *inprocess* action. The application of the corresponding change functions will determine subsequent dates of interest.

- The simulation system must always know what is the **next** date of interest of a *starting*, *restarting* or *inprocess* action. This date is always greater than (or equal to) the current date.
- For actions resulting in gradual changes, the **previous** date of interest is also useful. This date is always smaller than (or equal to) the current date.

In some sense, one can consider that the simulation system always needs to know an **interval of interest**, which is reduced to one date for actions resulting in instantaneous changes and which consists of two dates for actions resulting in gradual changes. If change functions allow to correctly update this information, the simulation system will at each step determine the smallest "next date of interest" and use the change function of the corresponding action to alter the representation of the world state and compute the next "next date of interest" of the action. This approach is close to that one of Hendrix [24] which consists of solving "sets of simultaneous equations to determine critical times in the set of ongoing processes". The main difference is that we explicitly decompose the resolution into (a) a domain-independent determination of the smallest "next date of interest" and (b) a domain-dependent computation of the next "next date of interest" of the corresponding action. In addition, the time variables associated with a process in [24] are "the time at which the process was initiated" and "the age of the process". We do not know the exact motivation for this choice and prefer the direct manipulation of "dates of interest".



```
[Action Description
    Action Type = release
    Parameters = ($object)
    Change Function = ●]

IF ($status = starting)
THEN IF (test (holds $agent $object))
        THEN (delete (holds $agent $object))
             (add (handempty $agent) (free $object))
             (set $status completed)
        ELSE (warning "action preconditions are not satisfied")
             (set $status completed)
ELSE (error "there is a bug in the simulation system")
```

Figure 6: A Change Function for Release Actions

Section 5.2 deals with the use of change functions to describe physical actions, section 5.3 with the use of change functions to describe cognitive actions, and section 5.4 with the description of interactions. These sections essentially consist of examples. They illustrate the application of the general approach outlined above.

5.2 Physical Actions

5.2.1 Instantaneous Actions

Figure 6 provides a change function for instantaneous *release* actions. Variables such as *$agent* and *$status* denote the values of the attributes (*agent* and *status*) of a particular action which is simulated (for example, the action *action-1* of figure 5).[4] The description is analogous to the description of an "operator" in a STRIPS-like planner [17]. If preconditions are satisfied (the agent holds the object), the operator is effectively applied. Some facts cease to be true (they are deleted) and some facts become true (they are added). If preconditions are not satisfied (the agent does not hold the object), a warning is issued and nothing changes. In both cases, the status of the action changes to *completed*. The action is considered *inprocess* for a period of time reduced to a single date. There is no future date of interest for a release action once the agent releases the object. Figure 7 shows the result of simulating the release action of figure 5. The set of propositions changes and the environment gets petrified since no agent performs other physical or cognitive actions.

5.2.2 Continuous Actions

Figure 8 provides a change function for *motion* actions. We suppose (a) that the *distance* function computes the distance between two locations and (b) that the agent moves at a constant speed. When the motion starts (or restarts), the location of the agent is set to

[4]We will also use *$next-date-of-interest* and *$previous-date-of-interest* in other examples.

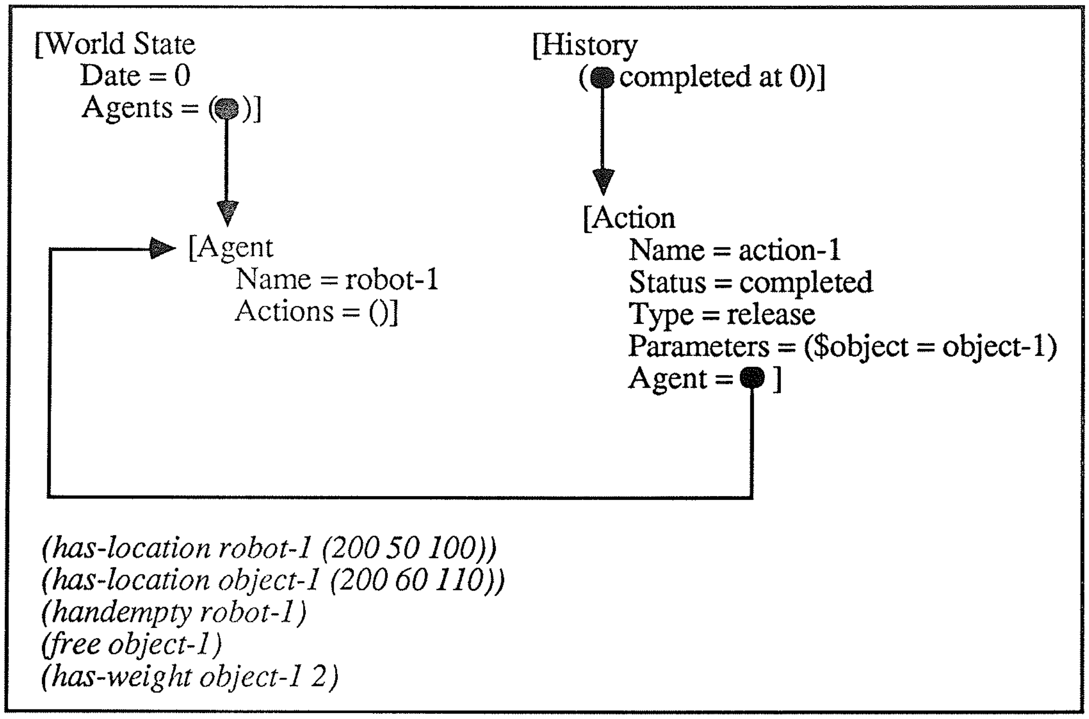


Figure 7: Simulating a Release Action

unknown and the next date of interest is incremented by the motion duration. Consequently, the second date of interest is when the agent reaches the goal. Then, the action ends. Note that an error will be detected if another change function attempts to determine the location of an agent while the agent moves. In this case, the simulation system will have detected an interaction unforeseen by the user. Figures 9 and 10 show the results of simulating the beginning and the end of the motion of figure 4. As previously, the environment gets petrified (even though the reasoning action is not *completed*).

If simulating the details of the motion is of some use, the user of the system can decompose the motion and allow the system to compute the motion from time t_i to time $t_{i+1} = t_i + \Delta$. One way to do this is to (a) define a function *motion-progress* which returns the new location of the agent after Δ units of time and (b) use the change function shown in figure 11. If the agent does not reach the goal, the next date of interest is incremented by Δ. Otherwise, the motion is *completed*. Let us note that the user can use any information in the *motion-progress* function, such as a path previously designed by the agent, variations of forces and velocities planned by the agent [2] or an "action vector" (*left-wheel-velocity right-wheel-velocity*) that the agent applies [28]. Similarly, the change function of figure 8 can be modified to compute the duration of the motion from such information.

The case of deviations to be expressed within the description of actions (cf. section 2.4) is quite important. In figure 12, we provide an example in which actual motion durations are randomly generated so that relative duration deviations satisfy a uniform distribution.

```
[Action Description
    Action Type = motion
    Parameters = ($goal)
    Change Function = ●]

IF ($status = starting) or ($status = restarting)
THEN (inc $next-date-of-interest
            (/ (distance (location $agent) $goal) (speed $agent)))
        (set (location $agent) unknown)
        (set $status inprocess)
ELSE IF ($status = inprocess)
        THEN (set (location $agent) $goal)
                (set $status completed)
        ELSE (error "there is a bug in the simulation system")
```

Figure 8: A Change Function for Motions

5.2.3 Interruptions

An action ends either when it is naturally completed or when another action provokes its interruption. In the second case, the action is *suspended* and a decision to resume the action is needed to restart it. In other terms, the resumption is part of another action which may occur sometime or never. To simulate the interruption of an action A, the user needs in some cases to provide, within the change function of the interrupting action, a way to characterize the state of the world in which the action A is interrupted. For example, if a motion corresponding to figure 8 is interrupted at time t, the change function of the interrupting action must include the computation of the location of the robot at time t. On the other hand, one can allow the interruption of a motion corresponding to figure 11 without worrying about anything provided that Δ is small enough (so that the error is negligible). Of course, an instantaneous action (e.g. figure 6) can never be *suspended* since it ends as soon as it starts.

5.2.4 Natural Processes and Unpredictable Events

The examples presented in the previous sections concern actions performed by agents operating in the simulated environment. Natural processes and unpredictable events are simulated the same way except that they are associated with fictive agents. Figure 13 provides a change function for the filling of a bucket. It is similar to the change function of figure 11. The function *content-progress* computes the new volume of water in the bucket with respect to any information about the tap and the bucket. The process ends either when the bucket is full or when water stops flowing from the tap.

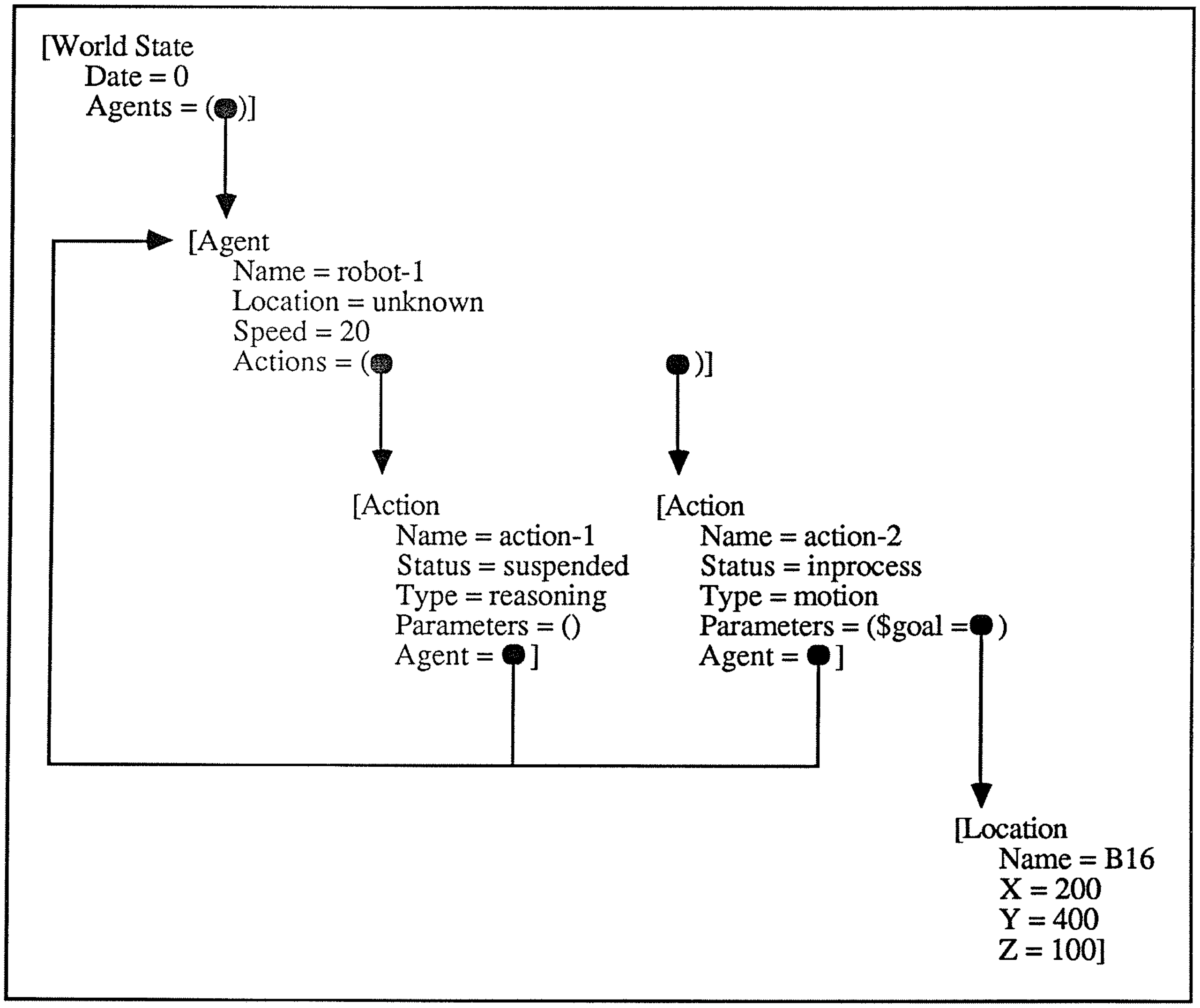


Figure 9: Simulating the Beginning of a Motion

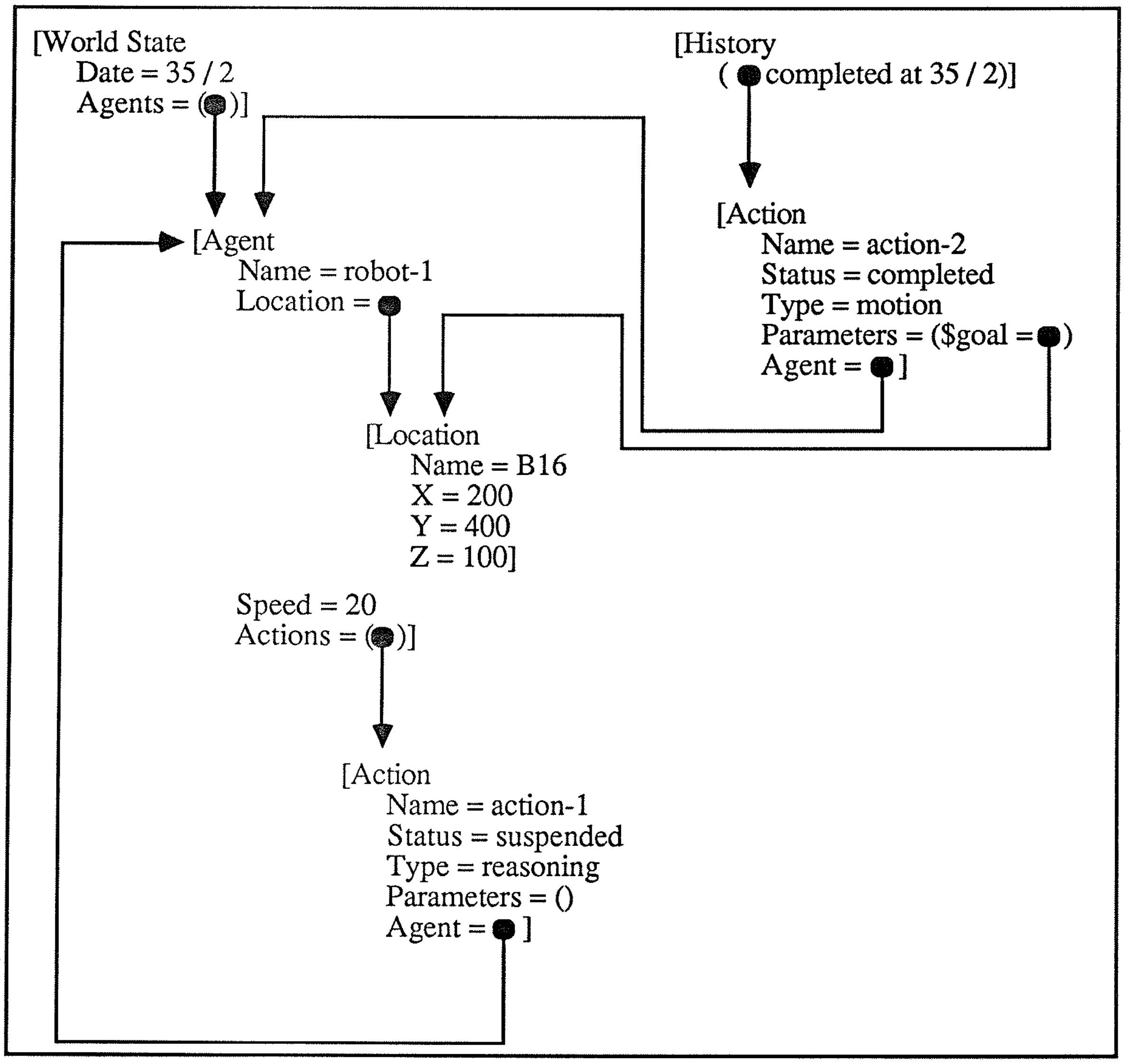


Figure 10: Simulating the End of a Motion

```
[Action Description
    Action Type = motion
    Parameters = ($goal)
    Change Function = ●]

IF ($status = starting) or ($status = restarting)
THEN (inc $next-date-of-interest Δ)
        (set $status inprocess)
ELSE IF ($status = inprocess)
        THEN (set (location $agent) (motion-progress $agent $goal Δ))
                IF ((location $agent) ≠ $goal)
                THEN (inc $next-date-of-interest Δ)
                ELSE (set $status completed)
        ELSE (error "there is a bug in the simulation system")
```

Figure 11: A Change Function for Motions

```
[Action Description
    Action Type = motion
    Parameters = ($goal)
    Change Function = ●]

IF ($status = starting) or ($status = restarting)
THEN (inc $next-date-of-interest
                (* (random-number (uniform-distribution 0.8 1.2))
                    (/ (distance (location $agent) $goal) (speed $agent))))
        (set (location $agent) unknown)
        (set $status inprocess)
ELSE IF ($status = inprocess)
        THEN (set (location $agent) $goal)
                (set $status completed)
        ELSE (error "there is a bug in the simulation system")
```

Figure 12: A Change Function for Motions with Duration Deviations

```
[Action Description
   Action Type = fill-bucket
   Parameters = ($bucket $tap)
   Change Function = ●]

IF ($status = starting)
THEN (inc $next-date-of-interest $\Delta$)
       (set $status inprocess)
ELSE IF ($status = inprocess)
      THEN (set (content $bucket) (content-progress $bucket $tap $\Delta$))
             IF ((content $bucket) $\neq$ (capacity $bucket)) AND ((flow-rate $tap) $\neq$ 0)
             THEN (inc $next-date-of-interest $\Delta$)
             ELSE (set $status completed)
      ELSE (error "there is a bug in the simulation system")
```

Figure 13: A Change Function for Filling Processes

Let us note that the change function does not tell when a filling process begins. This can appear as a problem, but in practice this problem allows many solutions. The solution we prefer consists of (a) associating preconditions with natural processes and (b) defining pattern-matching processes which initiate processes the preconditions of which become true. For example, "the bucket is under the tap", "the bucket is not full" and "water flows from the tap" are preconditions associated with a filling process. If one precondition becomes true, and the others are already true, then the fictive agent initiates a filling process. In some sense, the fictive agent executes pattern-matching processes which result in the initiation of other processes exactly as an actual agent performs cognitive actions which result in the initiation of other actions.[5]

5.3 Cognitive Actions

In this section, we will consider the description of cognitive actions in terms of change functions. Cognitive actions do not result in wide varieties of changes. Each of them affects the **memory** of the agent performing the action and, in some cases, provokes the initiation of other actions. The user of the system may use any formalism to represent the contents of the memory of an agent (e.g. logics of belief [25] [8] [38]) and distinguish between what is true in the environment and what each agent believes true. Let us note that this distinction does not necessarily result in duplications of information. For example, context mechanisms

[5]Another solution consists in defining change functions so that natural processes are always *inprocess*. For example, we can remove the instruction that sets the status to *completed* in figure 13 and declare filling processes *inprocess* in the initial state. When preconditions are not satisfied ("the bucket is not under the tap" or "the bucket is full" or "water does not flow"), the content of the bucket does not change, but the process continues. Let us note that this solution is not the most satisfactory: a great amount of computation is wasted when nothing changes. On the contrary, the amount of computation spent in pattern-matching increases with the amount of change. It is null when the environment is petrified.

(although the use of such mechanisms is costly [9]) and multiple-world knowledge representation systems [29] allow to represent the information available to each agent without enumerating all the epistemic propositions "agent believes proposition" which happen to be true. The user of the simulation system will often suppose that knowledge about unalterable well-known objects is inherited by some agents from the description of the environment.

The most important difficulties in simulating cognitive actions concern the relations between the cognitive actions of the same agent. It is very frequent for an agent to deal with several problems in parallel or to follow several lines of reasoning **in parallel**. The achievement of this macro-parallelism requires either to use several processors or to interleave different activities on the same processor. In both cases, control actions are necessary to manage dependencies between cognitive actions. Two cognitive actions are **independent** when they involve separate processors, access separate memory zones, and cannot interrupt each other. This is the easiest case to simulate. It is discussed in section 5.3.1. In the following sections, we consider the possible dependencies one after the other. We discuss the case in which several cognitive actions share memory zones in section 5.3.2, consider interruptions in section 5.3.3, and discuss the organization of various cognitive actions on the same processor in section 5.3.4.

5.3.1 Independent Cognitive Actions

In most cases, the cognitive actions of an agent are not independent. However, we will, in this section, present a change function for the simple case in which cognitive actions are independent. In the next sections, we will use this change function as a basis to investigate more interesting cases.

An independent cognitive action accesses either unalterable or "reserved" memory zones. By "reserved memory zones", we mean memory zones which are not accessed by other ongoing cognitive actions. An independent cognitive action may result in the alteration of the contents of reserved memory zones and in the initiation (or resumption) of other physical and cognitive actions. Its dates of interest are (a) its start time, (b) the dates at which it results in the initiation (or resumption) of other actions and (c) its end time. The goal of this section is to show how to define a change function which determines these dates of interest and simulates the corresponding initiation and resumption decisions.

We first derive a **computation function** from the procedure which embodies the cognitive action. The execution of the computation function consists of executing the procedure without actually initiating or resuming other actions. Instead, the computation function records in chronological order the initiations and resumptions that will occur. An additional attribute of the cognitive action is used for this purpose. We call this attribute *results* and use the notation *$results* to denote its value. Each element of *$results* is a triple (*date action status*) where *date* is the date at which the status of the action *action* is set to *status* (either *starting* or *restarting*). Another additional attribute is used to record the date at which the cognitive action ends. We call it *end-time* and use the notation *$end-time* to denote its value.

The main difficulty is to determine the dates to associate to each triple as well as the end time of the cognitive action. This requires the user to state assumptions about the speed of the computing system the agent uses to perform the action. This difficulty is unavoidable. When an extraordinary precision is required, its resolution can become very complex. However, requiring a great precision does not make sense if the time required for an agent to evaluate an instruction is allowed to fluctuate. Therefore, we distinguish two types of sensible cases:

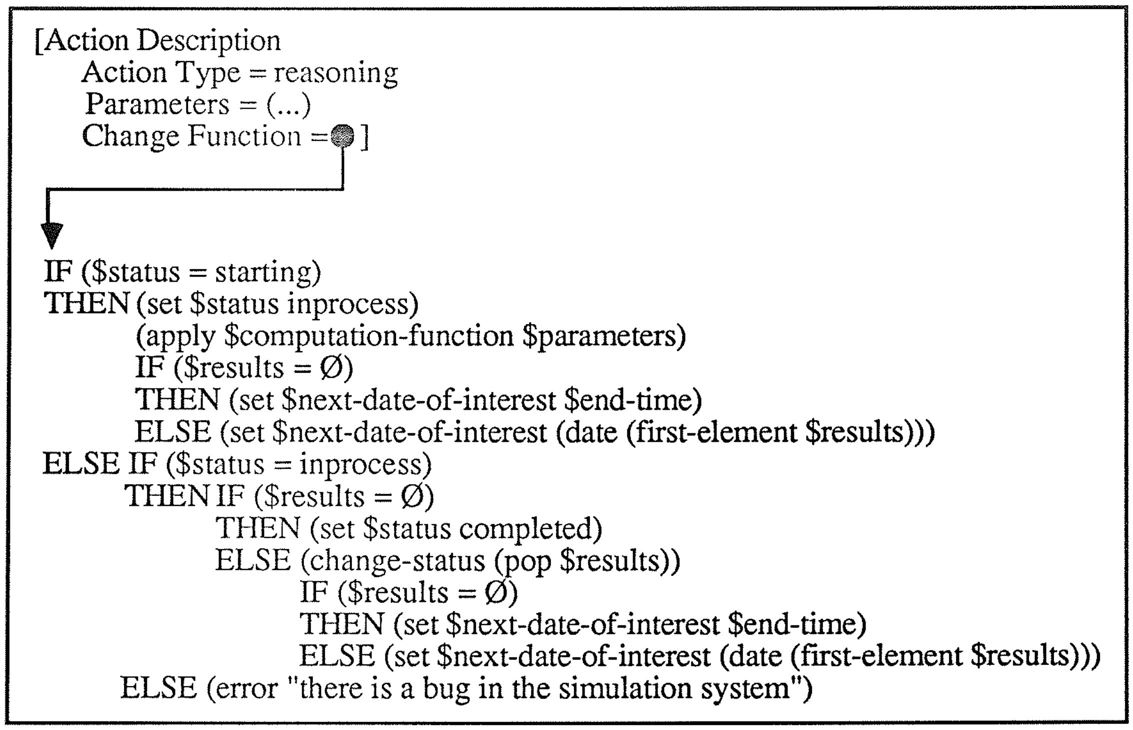
- In most cases which do not require a great precision, the user is able to relate the speed of the computing system the agent uses to perform the action to the speed of the computing system the simulation system uses to simulate the action. This means the user knows a function which satisfies the following property: when applied to the duration between (α) the beginning of the execution of the computation function and (β) the moment at which a triple is to be recorded in *\$results*, the function returns an acceptable estimate of the duration between (a) the beginning of the cognitive action and (b) the corresponding action initiation or resumption; when applied to the duration between (α) the beginning of the execution of the computation function and (γ) the moment at which the execution ends, the function returns an acceptable estimate of the duration between (a) the beginning of the cognitive action and (c) the end of the cognitive action. For example, if the computing system the simulation system uses is identical to the computing system the agent uses, the *identity* function satisfies these requirements.[6]

- If the time required for an agent to evaluate an instruction is known and not allowed to fluctuate, the duration between two instructions I and J is computable as the sum of the durations of the instructions executed between I and J. This solves the problem and provides the entire precision needed to simulate the actions of perfect real-time systems, the timing behavior of which is deterministic.[7]

Once the computation function is available, we use the change function of figure 14. When the cognitive action starts, the computation function is applied. This alters the contents of reserved memory zones and sets *\$results* and *\$end-time.* If *\$results* is not empty, the next date of interest is the date of the first triple. Otherwise, the next date of interest corresponds to the end of the cognitive action. In both cases, the change function is applied again at this date. If *\$results* is empty, the simulation of the cognitive action is completed. Otherwise, the first triple is extracted from *\$results.* The corresponding status change is made and the next date of interest of the cognitive action is set again. Eventually, *\$results* becomes empty and the status of the cognitive action becomes *completed.*

[6]Provided that paging and garbage-collecting activities are not too time-consuming. If this is not the case, the *identity* function tends to provide pessimistic estimates.

[7]When upper bounds are available instead of exact instruction durations, one can use them as pessimistic estimates. In fact, the resulting worst-case simulations are often the most interesting.



```
[Action Description
    Action Type = reasoning
    Parameters = (...)
    Change Function = ●]

IF ($status = starting)
THEN (set $status inprocess)
       (apply $computation-function $parameters)
       IF ($results = Ø)
       THEN (set $next-date-of-interest $end-time)
       ELSE (set $next-date-of-interest (date (first-element $results)))
ELSE IF ($status = inprocess)
      THEN IF ($results = Ø)
             THEN (set $status completed)
             ELSE (change-status (pop $results))
                    IF ($results = Ø)
                    THEN (set $next-date-of-interest $end-time)
                    ELSE (set $next-date-of-interest (date (first-element $results)))
      ELSE (error "there is a bug in the simulation system")
```

Figure 14: A Change Function for Independent Cognitive Actions

5.3.2 Cognitive Actions With Shared Memory

The case in which cognitive actions share memory is more complex. The simulation of these actions must accord with the order in which the corresponding procedures access and modify shared memory. In the worst case, one must decompose the simulation and simulate one instruction at a time. This is the only solution for cognitive actions which continually access and modify shared memory.

More economical solutions are practicable in cases in which shared memory is seldom accessed and modified (in cases in which the interactions between cognitive actions are few). We propose one which generalizes the change function presented earlier for independent cognitive actions. The rationale for the generalization consists of the three following statements:

- The additional dates of interest needed to perform a correct simulation correspond to the accesses and alterations of shared memory.
- The method used to manage status changes in section 5.3.1 is extendable to the management of alterations of shared memory. This means we modify the computation function so that it does not make the change. Instead, a triple (*date generalized-variable-to-set new-value*) is recorded in the *results* attribute of the cognitive action.
- A solution to execute an access at the right time *t* (once previous alterations have been executed) consists of (a) suspending the execution of the computation function prior to the access and (b) resuming the execution when the date is *t*. In other terms, the execution of the computation function is decomposed in "chunks" which begin with

shared memory accesses. Depending on the structure of the computation function, the more efficient manner to decompose its execution is either to insert interruptions within the function or to enable the system that interprets the function to detect the need to interrupt the execution. In both cases, the decomposition guarantees the correctness of the simulation.[8]

Figure 15 presents the new change function. A new attribute *stack* is used to know whether the execution of the computation function is completed (*$stack* is empty) or suspended prior to an access to shared memory (*$stack* describes the state in which the execution is suspended). In the first case, *$end-time* denotes the date at which the cognitive action ends. In the second case, it denotes the date of the memory access which caused the interruption (the end time of the previous "chunk"). When *$results* is empty and *$stack* is not empty, the execution of the computation function restarts. When *$results* is empty and *$stack* is empty, the cognitive action ends.

Let us note that the use of this change function incidentally allows to access the shared memory of an agent within the change function of a physical action. This represents the case in which an agent modifies some parameters of the physical action without considering the modification as the design of a new action. However, we did not meet any practical situation in which the management of a memory share between physical and cognitive change functions is more appropriate than the explicit design of new actions.

5.3.3 Interruptible Cognitive Actions

In this section, we consider the case of a cognitive action which provokes the interruption of another cognitive action. The simulation of such an interruption requires to reconsider the change functions of the two actions.

With respect to the change function of the interrupting action, what we did in section 5.3.1 about the initiation and resumption of actions applies equally well to the interruption of actions. The date of the interruption is a date of interest of the interrupting action and is recorded as such in the *results* attribute of the interrupting action.

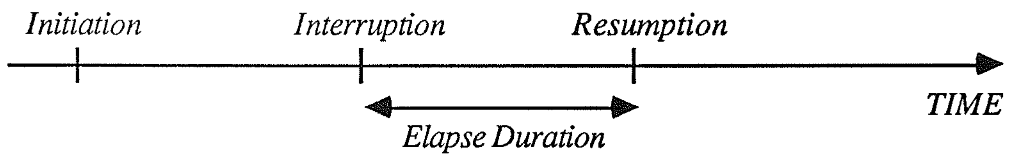
With respect to the change function of the interruptible action, the most important issue is the distinction between "shared" and "reserved" memory. The change function provided in section 5.3.2 is such that a part of the memory is "shared" as soon as two cognitive actions include — sometime between their initiation and their completion — modifications or accesses to this part of the memory. A given part of the memory is not "reserved" for a cognitive action A if another cognitive action accesses or modifies this part of the memory while A is *suspended.*

[8]One could also interrupt the execution of the computation function prior to status changes and alterations of shared memory. Then, the *results* attribute would not be needed. However, the management of interruptions is quite time-consuming at simulation time. We prefer to avoid them as much as possible.

```
[Action Description
    Action Type = reasoning
    Parameters = (...)
    Change Function = ●]

IF ($status = starting)
THEN (set $status inprocess)
        (apply $computation-function $parameters)
        IF ($results = Ø)
        THEN (set $next-date-of-interest $end-time)
        ELSE (set $next-date-of-interest (date (first-element $results)))
ELSE IF ($status = inprocess)
       THEN IF ($results = Ø)
               THEN IF ($stack = Ø)
                       THEN (set $status completed)
                       ELSE (resume $computation-function $parameters $stack)
                              IF ($results = Ø)
                              THEN (set $next-date-of-interest $end-time)
                              ELSE (set $next-date-of-interest (date (first-element $results)))
               ELSE (make-change (pop $results))
                      IF ($results = Ø)
                      THEN (set $next-date-of-interest $end-time)
                      ELSE (set $next-date-of-interest (date (first-element $results)))
       ELSE (error "there is a bug in the simulation system")
```

Figure 15: A Change Function to Manage Shared Memory



```
[Action Description
    Action Type = reasoning
    Parameters = (...)
    Change Function = ● ]

IF ($status = starting)
THEN (set $status inprocess)
        (apply $computation-function $parameters)
        IF ($results = Ø)
        THEN (set $next-date-of-interest $end-time)
        ELSE (set $next-date-of-interest (date (first-element $results)))
ELSE IF ($status = inprocess)
      THEN IF ($results = Ø)
              THEN IF ($stack = Ø)
                      THEN (set $status completed)
                      ELSE (resume $computation-function $parameters $stack)
                              IF ($results = Ø)
                              THEN (set $next-date-of-interest $end-time)
                              ELSE (set $next-date-of-interest (date (first-element $results)))
              ELSE (make-change (pop $results))
                      IF ($results = Ø)
                      THEN (set $next-date-of-interest $end-time)
                      ELSE (set $next-date-of-interest (date (first-element $results)))
   ELSE IF ($status = restarting)
         THEN (set $elapse-time (– $date $interruption-date))
                 (inc $end-time $elapse-time)
                 (for-each result in $results (inc (date result) $elapse-time))
                 (set $status inprocess)
                 IF ($results = Ø)
                 THEN (set $next-date-of-interest $end-time)
                 ELSE (set $next-date-of-interest (date (first-element $results)))
         ELSE (error "there is a bug in the simulation system")
```

Figure 16: A Change Function for Interruptible Cognitive Actions

As soon as the distinction between "shared" and "reserved" memory is clearly made, one just needs to consider the resumption of the interruptible action. Assuming that the time spent to restart the cognitive action is negligible, we use the date of the interruption (recorded at the time of the interruption) to compute the elapse duration between the interruption and the resumption. All the dates of interest that were previously computed (and stored within *$results* and *$end-time*) are increased accordingly. Figure 16 provides a pictural explanation of this process and presents the new change function. Of course, this change function is not always appropriate:

- As in section 5.3.2, simulating one instruction at a time is a better way to simulate cognitive actions which continually access and update shared memory.
- The time spent to restart a cognitive action is not always negligible. When very short cognitive actions are interruptible, one must take this time into account.

Let us note that the change function responds to all the difficulties discussed so far: initiation and resumption of other actions, cognitive actions with shared memory, interruption and resumption of interruptible actions. This is not always necessary. In the remainder of this section, we will discuss a case in which the interruption of the cognitive action is the sole concern. We suppose we are provided with (a) a planning algorithm which keeps improving a solution to a problem until an optimal solution is met and (b) a criterion for interrupting the execution of the algorithm. The regular evaluation of the criterion allows to make decisions between immediate action and further search for a better plan: when the evaluation returns *true*, the planning algorithm is stopped and the execution of the best available plan begins. An interesting example discussed in [4] concerns the traveling salesman problem [13] [39]. The problem is to find a tour as short as possible for a salesman (or a robot courier) which has to visit a specified list of locations and return to the original point of departure. An initial solution (if the salesman can go from any place to any other) is to visit the locations in the order in which they appear in the specified list. Then, each time the algorithm exhibits a tour shorter than the best available tour, the new tour becomes the best available tour and the algorithm continues, either until it is proven that the best available tour is the shortest possible, or until the agent decides to interrupt the algorithm execution.[9]

We suppose that the evaluation of the interruption criterion is performed outside of the planning algorithm. This allows the agent to interrupt the algorithm as soon as the criterion evaluates to *true*. It also allows the agent to allocate a processor to each of *n*

[9]The difficulty is of course to set the interruption criterion. One solution (proposed in [4] and [46]) is to use empirical knowledge obtained off-line, using statistical sampling methods. Shekhar and Dutta [42] prefer to provide a guarantee that the total (planning + execution) time is bounded by a function of the best obtainable (planning + execution) time (given the planning algorithm). In our understanding, none of the approaches proposed so far allows the agent to make simple decisions such as pursuing search for a while when a solution much better than previous solutions has been found. On the contrary, the criterion of Shekhar and Dutta tends to make an interruption follow the fortunate discovery of a very good solution. It does not allow to check whether there is an even better solution close to the good solution that was found.

```
[Action Description
    Action Type = reasoning
    Parameters = (...)
    Change Function = ● ]

 IF ($status = starting)
 THEN (set $status inprocess)
         (apply $computation-function $parameters)
         (set $next-date-of-interest $new-date)
 ELSE IF ($status = inprocess)
        THEN IF ($stack = Ø)
                THEN (set $next-date-of-interest ∞)
                ELSE (set $best-solution $new-solution)
                        (resume $computation-function $parameters $stack)
                        (set $next-date-of-interest $new-date)
        ELSE IF ($status = restarting)
                THEN (set $status inprocess)
                        (set $elapse-time (– $date $interruption-date))
                        (inc $new-date $elapse-time)
                        (set $next-date-of-interest $new-date)
                ELSE (error "there is a bug in the simulation system")
```

Figure 17: A Change Function for Anytime Algorithms

planning algorithms (or to the same algorithm working on different problems) and to keep one processor to determine when to interrupt each of these algorithms. The computation function corresponding to each algorithm results from the insertion of interruptions at those places in which new solutions — better than available solutions — are found. *$stack* describes the state in which the execution is suspended, *$new-solution* denotes the new solution and *$new-date* denotes the date at which this solution is available. *$next-date-of-interest* is set to *$new-date*. At this date, *$new-solution* becomes the best available solution (stored into *$best-solution*) and the execution of the computation function restarts to determine the next date of interest and solution. When the execution of the planning algorithm completes (*$best-solution* is *the* best solution according to the planning algorithm), *$stack* is set to $\emptyset$ and *$next-date-of-interest* to ∞.

Figure 17 provides the change function of the planning action. Interrupting this action consists in setting its status to *suspended* and getting *$best-solution* as the result of the interrupted algorithm. Let us note that this way to proceed is not too complex and guarantees a correct simulation of the agent's actions.

5.3.4 Operating Loops

In this section, we consider the situation in which an agent interleaves several cognitive activities on the same processor. We distinguish two cases:

- **Explicit control.** The control of the cognitive activities is explicit. This means that a set of policies provided by the designer of the agent (and maybe modified by the agent with respect to its current situation and problem-solving experience) determines which activity the agent pursues at any point in time. A consequence of this is that the cognitive activities do not need to be considered as separate actions. Rather, there is an overall cognitive process which consists of performing these activities and deciding how to interleave them.[10]

- **No explicit control.** The designer of the agent implements the execution of several cognitive actions as distinct processes without willing to investigate the exact behavior of the operating system interleaving these processes. A consequence of this is that there is no way to perform a very precise simulation of the behavior of the agent. But this does not matter much — at least for agents designed in computer science and robotics laboratories. Indeed, we cannot imagine situations in which the precision of the simulation is an issue while the control of the cognitive activities is not. When the timing behavior of the agent is crucial, the resolution of the control problem is crucial.

Let us first discuss the second case and devote the remainder of the section to an example in which control is explicit.

When there is no explicit control, the agent's behavior is approximated. The approximation depends on the knowledge that is available about the operating system interleaving processes. The easiest solution consists in using computation functions as in section 5.3.1 together with interactions as in (forthcoming) section 5.4.

- The computation function associated with a process P allows to determine what would be the next date of interest of P if P was the sole executing process.

- The change function associated with the interaction allows to gradually update such a date with respect to the context (number and characteristics of executing processes) and operating policy.

For example, let us suppose that computational time is equally distributed between executing processes and that computational time spent within the operating system to interleave processes is negligible. If *\$date* denotes the current date (after a date of interest of a process), if n denotes the number of processes executing at date *\$date* and if *min(\$next-date-of-interest)*

[10]In most cases, the control problem (deciding which activities to perform and when) is not solved on a dedicated processor. When it is, issues discussed in this section combine with issues discussed in sections 5.3.2 and 5.3.3.

denotes the smallest "next date of interest" over the n processes, the change function of the interaction adds $(n - 1)$ (*min($next-date-of-interest)* $-$ *$date*) to each "next date of interest". If the set of processes does not change prior to the new *min($next-date-of-interest)*, the change function of the corresponding process is applied at this date. Then, the change function of the interaction is applied again. If the set of processes changes prior to the new *min($next-date-of-interest)* (e.g. a new process starts), the change function of the interaction modifies the dates of interest to account for the change.

To discuss the case in which control is explicit, we consider a simple event-driven system inspired by the BB1 control architecture [22]. Each cognitive activity is implemented as a set of **knowledge sources** to which both triggering conditions and execution preconditions are attached. Significant modifications of the robot memory are called **events.** Each event can trigger knowledge sources and each triggered knowledge source gives rise to a **ksar** (knowledge source activation record) available for execution as soon as its preconditions are satisfied. The basic control loop that runs the whole system (figure 18 (from [11])) consists of the following steps:

- Update an agenda of pending reasoning actions (ksars), according to the arrival of new events.
- Choose a ksar in the agenda with respect to a set of control policies.
- Execute the selected ksar.

The execution of a reasoning action produces new events. These events are considered on the next cycle. In addition, the system uses a mailbox to receive events from other processors. This mailbox is read and reset at each cycle — when the first step begins. Consequently, the dates of interest of the process are the dates at which new cycles begin and the dates at which the execution of chosen reasoning actions results in the initiation (or interruption or resumption) of other actions or in the access (or alteration) of memory zones shared with other processors. The general case is consequently modelled as in section 5.3.2 (with the same change function). In more specific cases, we can use simpler change functions. For example, figure 19 presents the change function we use when the mailbox is the unique shared memory zone. In this case, we (1) derive a computation function which stops at the end of each cycle and sets the *results* attribute as in section 5.3.1 and (2) modify the change function proposed in section 5.3.1 to execute a new cycle when *results* is empty. The resulting change function allows to simulate the overall cognitive process in a very efficient fashion.

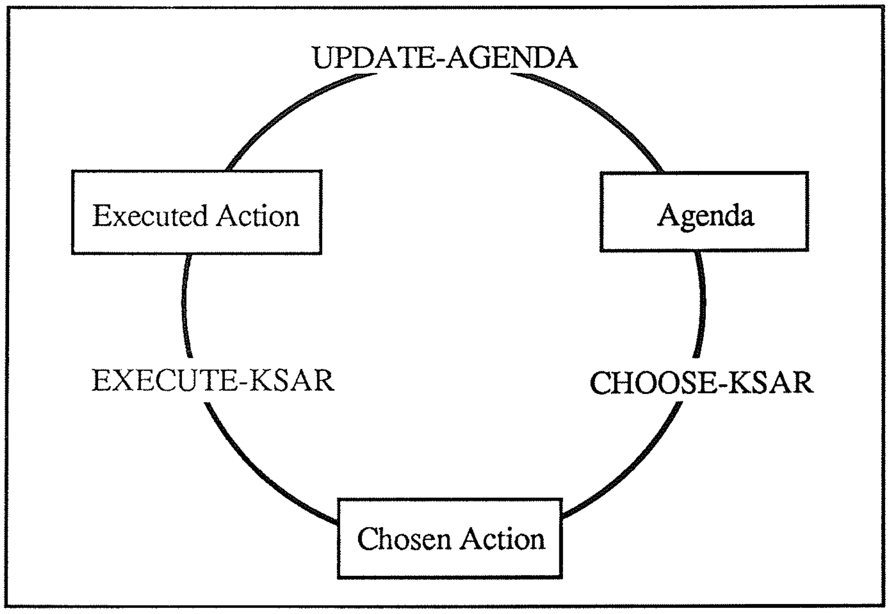
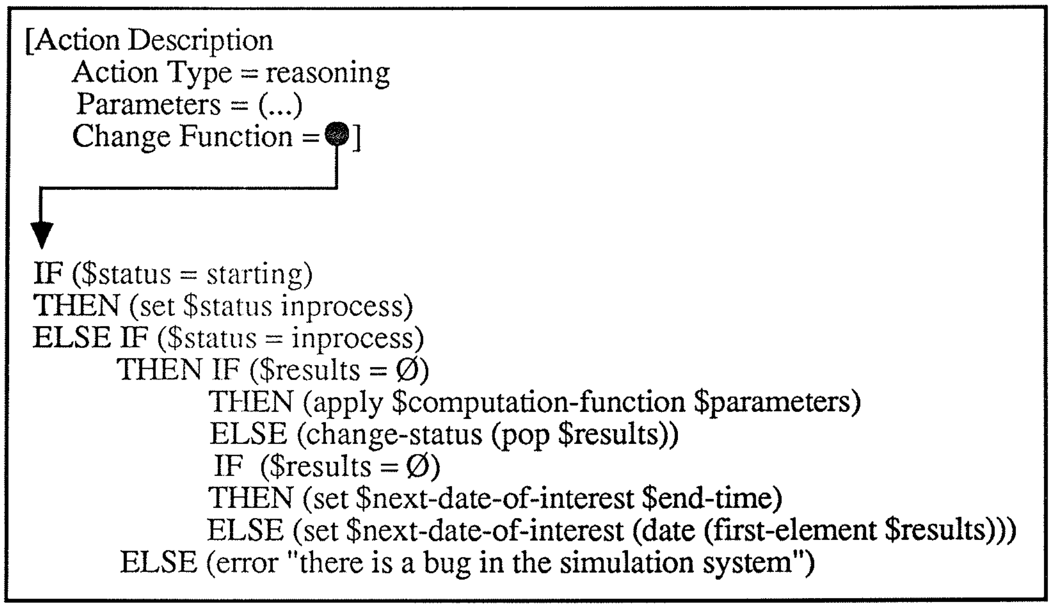


Figure 18: The BB1 Basic Control Loop



```
[Action Description
    Action Type = reasoning
    Parameters = (...)
    Change Function = ●]

IF ($status = starting)
THEN (set $status inprocess)
ELSE IF ($status = inprocess)
        THEN IF ($results = Ø)
                THEN (apply $computation-function $parameters)
                ELSE (change-status (pop $results))
                IF  ($results = Ø)
                THEN (set $next-date-of-interest $end-time)
                ELSE (set $next-date-of-interest (date (first-element $results)))
        ELSE (error "there is a bug in the simulation system")
```

Figure 19: A Change Function for Operating Loops (the computation function corresponds to one cycle)

5.4 Interactions

At some levels of abstraction, the effects of several concurrent actions often differ from the effects of these actions considered separately. In such a case, the user of the simulation system can provide a more detailed description of each action. However, the description of an explicit interaction is in some cases a more efficient solution. For example, let us consider a (well-known) footrace between Achilles and a tortoise (which starts some distance ahead) and imagine that the tortoise gives up and stops as soon as Achilles passes it. To simulate a race, we need to compute when Achilles catches up with the tortoise. If an approximate simulation is acceptable, the system user may discretize motions with more or less precision. However, another possibility is to describe a "pass" interaction between the two motions.

For each type of interaction, the user of the simulation system provides a change function and a way to determine interaction start times. In our example, a function of the two motions (characterized by initial locations, start times and agent velocities) is needed to compute the date of the interaction. Whenever two motions are executed in parallel, this function is applied to determine whether there will be an interaction and when. The computed start time is the first date of interest of the interaction. The change function is applied at this date. The position of the tortoise is computed and the status of the tortoise's motion is set to *suspended* (or *completed*).

Figure 20 presents a slightly different description of the interaction. The effect of the interaction is that the two agents "know" that they have reached the same location (in a 1D space). Whether one agent stops or not depends on the cognitive actions that this event triggers.

The *simulation* function is designed to ensure that an interaction occurs only if the corresponding actions continue until the interaction start time. Consequently, the correctness of the simulation essentially depends on the computation of interaction start times. The representation of an interaction is not suitable when the system user does not know how to compute its start time.

For example, some approximations of the "pass" interaction conduct to infinite loops in accordance with Zeno's paradox (see [26]). These approximations are not appropriate if the actual event of interest is when Achilles passes the tortoise.[11]

[11]They are appropriate if the user is interested by the dates at which Achilles reaches the previous position of the tortoise. There is no need to disallow infinite loops involving bounded series of dates. From a phenomenological point of view, we can even observe that the time "sensed" by an observer of the simulation system corresponds to the time a theoretical observer would sense if he was directly observing Achilles and the tortoise — and reacting to the simulated events (and only to those events). For such an observer, the race would actually never end.

```
[Interaction Description
    Interaction Type = pass
    Parameters = ($action-1 $action-2)
    Date Function = ●

 IF ((type $action-1) = motion) and ((type $action-2) = motion)
 THEN (set $agent-1 (agent $action-1))
         (set $agent-2 (agent $action-2))
         (set $date-1 (previous-date-of-interest $action-1))
         (set $date-2 (previous-date-of-interest $action-2))
         IF (solve-constraints
                 ($t)
                 (= (+ (location $agent-1) (* (speed $agent-1) (– $t $date-1)))
                     (+ (location $agent-2) (* (speed $agent-2) (– $t $date-2))))
                 (≤ $date $t))
         THEN (return $t)
         ELSE (return ∞)
 ELSE (return ∞)

    Change Function = ●]

 IF ($status = starting)
 THEN (set $agent-1 (agent $action-1))
         (set $agent-2 (agent $action-2))
         (write-in-mailbox $agent-1 (same-location $agent-1 $agent-2))
         (write-in-mailbox $agent-2 (same-location $agent-1 $agent-2))
 ELSE (error "there is a bug in the simulation system")
```

Figure 20: An Interaction

6 Event-Driven Simulations

In section 5, we have discussed the representation of actions and interactions in terms of change functions. Given these change functions, the evolution of an environment is simulated as a discrete succession of events. Time is not incremented in units. Scheduled events are gathered in a list and time is, at each simulation cycle, incremented to the date of the next scheduled event. The *simulation* function consists of a three-step simulation loop:

- *Compute a list of ongoing actions and forthcoming interactions.* For each agent, the ***actions*** attribute provides the set of actions performed by the agent. Interactions are determined from the overall set of actions as explained in section 5.4.

- *Determine actions and interactions for which the next date of interest is minimal.* Only interactions are retained when actions and interactions share the same "next date of interest".

- *Alter the representation of the environment in accordance with the change functions of the selected actions (or interactions).* When several actions share the same "next date of interest", change functions are applied for all of these actions. The change functions are supposed to commute. When two change functions do not commute, there is an interaction between the corresponding actions.

The simulation is pursued either for a given period of time or until a ***stop-test*** function returns ***true*** (cf. section 3). In both cases, continuous actions require additional simulation steps:

- If the evolution of the environment is simulated until the date equals ***date***, ***date*** is a date of interest of the continuous actions which start before ***date*** and end after ***date***. When all the dates of interest computed by change functions are greater than ***date***, the simulation system sets all the next dates of interest of continuous actions to ***date*** and applies the corresponding change functions prior to stop.

- Similarly, the date at which ***stop-test*** becomes true is a date of interest of the continuous actions which start before this date and end after this date. When ***stop-test*** becomes true, the simulation system sets all the next dates of interest of continuous actions to the current date and applies the corresponding change functions prior to stop.[12]

When the user of the system does not specify a period of time or a ***stop-test*** to stop the simulation, the system runs indefinitely or stops when the environment is petrified, in a state with no ongoing action.

[12]The date at which ***stop-test*** becomes true is also a date of interest of the action which makes it true. For a continuous action, this date cannot be computed "exactly" if the change function (provided by the user) does not perform this computation.

7 Conclusion

In this paper, we have described a simulation system specially designed to simulate actions of autonomous agents. An important characteristic of this system is that it allows its users to use any formalism to describe the simulated environment and the actions performed in this environment. Only a minimal amount of information must be encoded within the object-oriented representation framework of the simulation system. The system needs to know the date, the set of existing agents and the set of actions performed by each agent in the current state.

- The simulation system allows the user to choose any totally ordered set S as a model of time. Dates are members of S and the user provides a definition of the ordering relation in the form of two functions *time*= and *time*$\geq$. To manipulate the concept of duration (which is not compulsory), the user provides an additional function *time*+ allowing to add dates and durations. This is similar in spirit to the modelling flexibility discussed in [35]. This flexibility is particularly important when dealing with time because the concept of *time* covers very different representations, which are as many ways of considering sequences of events [3]. There is no reason to enforce a particular representation. The most adequate representation varies from one application to another.

- Two attributes (*name* and *actions*) are needed to describe an agent. The other characteristics of an agent are not directly accessed by the *simulation* function. They are application-dependent and the user can use any formalism to represent them.

- Similarly, five attributes (*name*, *agent*, *status*, *type* and *parameters*) are needed to describe an action. For each type of action, the user provides a change function which describes the actions with the precision needed for the considered application. Consequently, compromises between efficiency and precision are made by system users: the system does not detail the simulation of an action when the user considers the details are not important.

Many types of actions can be represented precisely enough to perform correct simulations.

- The system allows the simulation of cognitive actions. The representation of cognitive actions is similar to the representation of physical actions (e.g. motions). Both the computations and the evolution of the environment are simulated: there is no a priori distinction between cognitive and physical actions. Section 5.3 provides many examples of cognitive actions easily simulated by the system. This includes the use of *anytime algorithms* [4] [14] and the organization of various cognitive actions within a *blackboard system* [22].

- The system allows the description of instantaneous events, discrete actions and continuous processes. To deal with continuous processes, the user may (a) either supply an adequate time increment allowing to discretize the process or (b) provide functions to solve sets of exact equations representing the process (as in [24]). This representation alternative allows the user to choose an appropriate level of abstraction for each physical and cognitive action.
- The system does not only simulate "actions", each of which is performed by an "agent". It also allows the representation and the simulation of interactions between concurrent actions, predictable and unpredictable events occurring without involving agents (e.g. natural effects that do not directly follow an action), deviations from the normal course of an action (e.g. statistical distribution of the action duration) and interruptions.

Some of the issues discussed in this paper have already been considered by other researchers. In particular, the systems presented in [20] and [24] share important characteristics with our simulation system. The MACE environment [20] has been designed to serve as a testbed for experimenting with a variety of DAI (Distributed Artificial Intelligence) systems. MACE agents are essentially computational units performing cognitive actions in parallel. While [20] focuses on DAI, we propose a simulation system allowing to make experiments with all sorts of agents evolving in all sorts of environments. In our system, all the physical and cognitive capabilities of an actual agent (sensing, reasoning, moving) are addressed. The relation with the work of Hendrix [24] is also interesting. Hendrix notices that the evolution of the world is not always describable as a succession of discrete operators. Consequently, he proposes a system to represent and simulate simultaneous actions and continuous processes. This system is used by a robot to determine which courses of actions permit to meet a given goal. We believe this system can be extended to represent cognitive actions and serve as a simulation system per se. Similarly, we could provide a robot with our simulation system and let the robot perform simulations to test its plans (e.g. in order to detect subtle interactions unforeseen during planning). In this respect, let us note that the similarity between the simulation algorithm (cf. section 6) and operating loops defined in section 5.3.4 allows to implement each change function as an independent "knowledge source". When this is done, the performance of a *simulation* is defined as the performance of a succession of cognitive actions. We can easily simulate agents performing simulations.

The simulation system is still experimental. Nevertheless, we believe it is the most practical to test and compare reactive system designs. We are currently considering applications in various types of environments: office environments, shop-floors, construction sites. The applications and the types of experiments we plan to perform are detailed below. In addition to allowing correct experiments, the simulation system also simplifies the implementation of distributed problem-solvers. For example, we have made a new version of a distributed planner originally developed at University Paris VI [16]: MASH (Multi-Agent Satisfaction Handler). The implementation took only fifteen hours (for the MASH system + an application + functions to generate examples + debugging) and the resulting system runs nearly as fast as a COMMON-LISP version of the original MASH system (in the worst case, 36 milliseconds against 25 milliseconds for the construction of a "FRUITCAKE" tower (see [16]) on a DEC station 3100). The rapidity of the implementation is probably due to the clarity of the concepts in [16]. Nevertheless, the use of the simulation system allows to encode the behaviors of each agent with no unnecessary worry about the coordination of the overall society. This definitely saves time.

Office Automation

Currently, the most advanced application concerns the combination of centralized and distributed multi-robot planning and scheduling techniques in an office environment [6] [36]. Typical tasks include transportation of objects (books, mail), operation of machines (copiers, vending machines), cleaning and maintenance. Different communication systems (necessary to coordinate operations and for each robot to acquire information) may be available. Communications may be more or less costly, always possible or possible only at some moments. This suggests various system organizations which are as many intermediates from a completely centralized system to a completely distributed system. In this context, the simulation system is used to test and compare various combinations of centralized and distributed problem-solving components (task and motion planning systems, task allocation systems, execution control systems) integrated (for each agent) within the blackboard architecture presented in section 5.3.4. In the near future, we will conduct experiments to determine to what extent a centralized analysis of task interactions allows to improve the overall behavior of the robotic system. We could also determine to what extent task allocation is enhanced when details about the environment geometry are taken into account. Later, we can also use the simulation system to:

- examine the problems which arise when a robot takes too much time to revise its plans;
- check the utility of different sensors and interpretation algorithms;
- test contingency-tolerant motion execution techniques involving artificial potential fields (see [31] [7]);
- test and compare methods allowing robots to determine (in a given situation) which pieces of information are worth getting (or spreading);[13]
- compare the blackboard architecture currently in use with other agent architectures (e.g. action networks [41]) in terms of *speed* and *responsiveness* (and possibly other criteria such as those defined in [15] and [12]).

[13]This includes the comparison of mechanisms allowing error error propagation to stop: unless each robot carefully verifies critical pieces of information prior to propagate them, we can expect errors concerning important issues to propagate much quicker than less important errors (as in the case of human rumors [1]). Error propagation is therefore an important problem in domains in which important errors can be made.

Manufacturing (In Clean Rooms)

The production of microelectronic circuits in clean rooms raises various organization and scheduling problems.[14] First, the cost of a clean room (typically $10M) increases quite rapidly with the size of the clean room. Hence a need to automate manufacturing operations and part transportations, in order to allow many parallel actions in the smallest amount of space. The risk of wafer contamination also suggests the avoidance of manual operations. In particular, using mobile robots to transport circuits (in protective boxes) from one small clean room to another is probably safer than manually transporting wafers in a large clean room. This leads to the exploration of issues similar to those considered in the office automation project. In addition, unpredictability in factory operation is particularly important: the methodology

[14]We are going to use a few technical terms. Let us consequently draw a simple picture of circuit fabrication for people to which this process is not familiar. Circuits are produced in *wafers*: a few hundreds of identical circuits are usually grown on the same wafer. When the manufacturing process begins, a wafer is a thin polished slice of silicon. Circuits are then constructed in layers. Each layer is defined by a pattern on a glass plate, called a *photomask*, whose features are transferred to the surface of the wafer. The process consists, for each layer, of a varying combination of *deposition*, *photolithographic* and *etching* steps.

- *Deposition.* Material is either grown on the surface of the wafer or implanted within the wafer at a pre-determined depth. Many types of deposition processes are distinguished. *Oxidation* consists of heating the wafer in an atmosphere of oxygen and water vapor so that a film of silicon dioxide forms on the surface of the wafer. Silicon dioxide subsequently protects the wafer from its environment and insures electrical insulation between metals. *Metallization* is a process in which various regions of the device structure are connected with evaporated aluminium which deposits on the wafer to produce a circuit. *Implantation* consists of accelerating a controlled amount of selected particles (e.g. boron ions), so that they penetrate below thin layers of oxide, but not below thicker layers. *Photoresist application* consists of placing a drop of photoresist (a substance sensitive to ultra-violet radiation but not to yellow light) dissolved in a solvent on the wafer, rapidly spinning the wafer to obtain a thin film, and baking the wafer to increase adherence.
- *Photolithography.* Ultra-violet radiation allows to transfer patterns from a photomask to the surface of the wafer: photoresist is attacked in the areas exposed to the radiation. A new bake (at a very high temperature) usually follows this exposition to harden the remaining resist. The most important difficulty in photolithography is to align the photomask to any previously defined pattern on the wafer. Aligners are very precise and consequently expensive pieces of equipment. As a result, photolithography is often a bottleneck operation.
- *Etching.* Acids or magnetic fields are used to attack either the portions of the wafer which are not protected by the hardened resist or (later, in some cases after an intermediate implantation step) the resist itself.

In addition, the manufacturing process contains many cleaning and inspection steps, not only at the end of the process (where tests are made to mark defective circuits prior to section the wafer), but also between deposition, photolithographic and etching steps. An important characteristic of the overall process is that the path that a wafer follows varies with the layer and with inspection results. Therefore, the factory cannot be organized as a production line. It is rather a collection of production sectors among which wafers travel in a complex fashion. Another important characteristic is the risk of *wafer contamination* (introduction of unwanted impurities within the circuits) especially during metallization and implantation steps. To reduce this risk, manufacturing operations are performed in *clean rooms* with refreshed air and human operators provided with special equipment.

specifies cases in which the path followed by a set of circuits depends on test and inspection results.[15] Consequently, it does not make sense to generate a precise production schedule for a week or a month. Instead we would rather (a) use probabilistic or fuzzy models to generate long-term schedules [40] [30], (b) generate more precise schedules for short periods of time (not more than a day) and (c) use a very reactive scheduling system to update the immediate part of the schedule as unexpected events occur. More specifically, we are now investigating the relation between the short-term predictive scheduling system and the reactive scheduling system.

- The predictive scheduling system schedules operations *in advance* (typically in the evening for the next day). A predictive scheduling system is not absolutely necessary, but its usefulness is usually significant. Because it is not subjected to real-time constraints, computational time can be spent to ensure the global quality of the job-shop schedule. Furthermore, the existence of a predictive schedule allows to prepare tools, make transportations and perform set-ups *in advance*. Nevertheless, the uncertainty in the environment suggests the maintenance of many ordering possibilities allowing the reactive system to cope with unexpected events during execution: as mentioned in [44], the precision of schedules must be determined by the uncertainty of the information used in making decisions (see also [18] [19] [33]).

- The reactive scheduling system makes scheduling decisions in real-time with respect to the actual state of the shop. It modifies the predictive schedule in response to unexpected events which modify the shop conditions. An important issue is therefore to determine what is a good horizon for the reactive scheduling system: is it allowed to modify whatever it wants in the schedule or should it restrict its modifications to a few coming hours ? This is particularly important because the reduction of the reactive system's horizon does not only allow a better preservation of the schedule. It also ensures a prompt reaction. In a typical circuit factory, 1000 lots and 100 agents (human operators, machines, robots) are considered at any instant in time. This is much more than in the cases discussed in section 2.1 and suggests the use of more restricted (less complete) reactive strategies. Another possibility of interest is to distribute the reactive scheduling effort: for example, we could associate a distinct reactive scheduling agent to each production sector.

- Although the reactive system must always go on making decisions in real-time, it may in some cases alert the predictive system that some manufacturing operations are not executed as scheduled. The predictive system would then re-schedule some operations (which are not imminent) with a more global view. This requires a criterion to determine when it is suitable to re-schedule operations with this more global view.

[15]If we except this, very few unpredictable events such as machine breakdowns (except simple problems with aligners) occur. In other terms, unpredictability arises mostly from the existing methodology and not from the complexity of the environment.

The simulation system will certainly prove useful to make experiments involving various versions of predictive and reactive scheduling systems. In particular, we may partly re-use the model built as part of the office automation project to compare more or less distributed versions of the reactive scheduling system.

Automation on a Construction Site

The problems we consider in the case of a construction site are similar to the problems relating to office automation. We are investigating the integration of various short-term planning and execution monitoring techniques on a construction site. The case of a construction site introduces particular difficulties. For example, the geometry of a construction site continually changes while an approximate map of an office environment is generally available. More sophisticated planning techniques (integrating temporal and geometrical reasoning) are consequently necessary. For the same reason, there are many more real-time interactions between agents. In this context, the definition of an efficient and correct experimental approach to compare reactive system designs is a critical issue. We believe the simulation system described in this paper will allow to perform valid experiments in an efficient fashion.

References

[1] Gordon W. Allport and Leo Postman. *The Psychology of Rumor.* Russell and Russell, 1965.

[2] Jérôme Barraquand, Bruno Langlois and Jean-Claude Latombe. *Robot Motion Planning With Many Degrees of Freedom and Dynamic Constraints.* Proceedings of the Fifth International Symposium on Robotics Research, Tokyo, Japan, 1989.

[3] Emile Benveniste. *Problèmes de linguistique générale II.* Gallimard, 1974 (in French).

[4] Mark Boddy and Thomas Dean. *Solving Time-Dependent Planning Problems.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, 1989.

[5] Frederick P. Brooks Jr. *The Mythical Man-Month. Essays on Software Engineering.* Addison-Wesley, 1975.

[6] Philippe Caloud, Wonyun Choi, Jean-Claude Latombe, Claude Le Pape and Mark Yim. *Indoor Automation With Many Mobile Robots.* Proceedings of the IEEE International Workshop on Intelligent Robots and Systems, Tsuchiura, Japan, 1990.

[7] Wonyun Choi, David Zhu and Jean-Claude Latombe. *Contingency-Tolerant Robot Motion Planning and Control.* Proceedings of the IEEE International Workshop on Intelligent Robots and Systems, Tsukuba, Japan, 1989.

[8] Philip R. Cohen and C. Raymond Perrault. *Elements of a Plan-Based Theory of Speech Acts.* Cognitive Science, 3(3):177-212, 1979.

[9] Anne Collinot. *ART : un premier bilan.* Rapport technique, Laboratoires de Marcoussis, 1986 (in French).

[10] Anne Collinot et Claude Le Pape. *Comparaison de plusieurs modes d'utilisation d'un système d'ordonnancement flexible.* Rapport technique, Laboratoires de Marcoussis, 1988 (in French).

[11] Anne Collinot. *Un cycle d'exécution utilisant des connaissances heuristiques pour satisfaire des contraintes de temps réel.* Rapport intermédiaire de bourse INRIA, Stanford University, 1989 (in French).

[12] Anne Collinot and Barbara Hayes-Roth. *Real-Time Control of Reasoning: Experiments with Two Control Models.* Technical Report, Stanford University, 1990.

[13] G. Dantzig, R. Fulkerson and S. Johnson. *Solution of a Large-Scale Traveling Salesman Problem.* Operations Research, 2(4):393-410, 1954.

[14] Thomas Dean and Mark Boddy. *An Analysis of Time-Dependent Planning.* Proceedings of the Seventh National Conference on Artificial Intelligence, Saint Paul, Minnesota, 1988.

[15] Rajendra Dodhiawala, N. S. Sridharan, Peter Raulefs and Cynthia Pickering. *Real-Time AI Systems: A Definition and An Architecture.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, 1989.

[16] Jacques Ferber and Eric Jacopin. *A Multi-Agent Satisfaction Planner for Building Plans as Side Effects.* Technical Report, University Paris VI, 1990.

[17] Richard E. Fikes and Nils J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* Artificial Intelligence, 2(3):189-208, 1971.

[18] Barry R. Fox and Karl G. Kempf. *Opportunistic Scheduling for Robotic Assembly.* Proceedings of the IEEE International Conference on Robotics and Automation, Saint Louis, Missouri, 1985.

[19] Barry R. Fox and Karl G. Kempf. *Reasoning about Opportunistic Schedules.* Proceedings of the IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.

[20] Les Gasser, Carl Braganza and Nava Herman. *Implementing Distributed AI Systems Using MACE.* Proceedings of the Third Conference on Artificial Intelligence Applications, Kissimmee, Florida, 1987.

[21] Michael P. Georgeff and Amy L. Lansky. *Reactive Reasoning and Planning.* Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, Washington, 1987.

[22] Barbara Hayes-Roth. *A Blackboard Architecture for Control.* Artificial Intelligence, 26(3):251-321, 1985.

[23] Barbara Hayes-Roth. *A Multi-Processor Interrupt-Driven Architecture for Adaptive Intelligent Systems.* Technical Report, Stanford University, 1987.

[24] Gary G. Hendrix. *Modelling Simultaneous Actions and Continuous Processes.* Artificial Intelligence, 4(3):145-180, 1973.

[25] Jaakko Hintikka. *Knowledge and Belief. An Introduction to the Logic of the Two Notions.* Cornell University Press, 1962.

[26] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Vintage Books, 1980.

[27] Vasudevan Jagannathan, Rajendra T. Dodhiawala and Lawrence S. Baum (editors). *Blackboard Architectures and Applications.* Academic Press, 1989.

[28] Leslie Pack Kaelbling. *Goals as Parallel Program Specifications.* Proceedings of the Seventh National Conference on Artificial Intelligence, Saint Paul, Minnesota, 1988.

[29] Hervé Kauffmann and Alain Grumbach. *Representing and Manipulating Knowledge Within "Worlds".* Proceedings of the First International Conference on Expert Database Systems, Charleston, South Carolina, 1986.

[30] R. M. Kerr and R. N. Walker. *A Job-Shop Scheduling System Based on Fuzzy Arithmetic.* Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production Planning and Control, Charleston, South Carolina, 1989.

[31] Oussama Khatib. *Real-Time Obstacle Avoidance for Robot Manipulators and Mobile Robots.* International Journal of Robotics Research, 5(1):90-98, 1986.

[32] Thomas J. Laffey, Preston A. Cox, James L. Schmidt, Simon M. Kao and Jackson Y. Read. *Real-Time Knowledge-Based Systems.* AI Magazine, 9(1):27-45, 1988.

[33] Claude Le Pape and Stephen F. Smith. *Management of Temporal Constraints for Factory Scheduling.* Technical Report, Carnegie-Mellon University, 1987.

[34] Claude Le Pape. *Des systèmes d'ordonnancement flexibles et opportunistes.* Thèse d'Université, Université Paris XI, 1988 (in French).

[35] Claude Le Pape. *The Use of a Flexible Constraint Propagation System.* Proceedings of the AAAI Workshop on Constraint Processing, IJCAI, Detroit, Michigan, 1989.

[36] Claude Le Pape. *A Combination of Centralized and Distributed Methods for Multi-Agent Planning and Scheduling.* Proceedings of the IEEE International Conference on Robotics and Automation, Cincinnati, Ohio, 1990.

[37] Claude Le Pape. *Simulating Actions of Autonomous Agents: An Overview.* Proceedings of the AAAI Workshop on Artificial Intelligence and Simulation, AAAI, Boston, Massachusetts, 1990.

[38] Hector J. Levesque. *A Logic of Implicit and Explicit Belief.* Proceedings of the Fourth National Conference on Artificial Intelligence, Austin, Texas, 1984.

[39] John D. C. Little, Katta G. Murty, Dura W. Sweeney and Caroline Karel. *An Algorithm for the Traveling Salesman Problem.* Operations Research, 11(6):972-989, 1963.

[40] Nicola Muscettola and Stephen F. Smith. *A Probabilistic Framework for Resource-Constrained Multi-Agent Planning.* Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, 1987.

[41] Nils J. Nilsson. *Action Networks.* Proceedings of the Rochester Planning Workshop "From Formal Systems to Practical Systems", Rochester, New York, 1988.

[42] Shashi Shekhar and Soumitra Dutta. *Minimizing Response Times in Real-Time Planning and Search.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, 1989.

[43] Aaron Sloman. *Real-Time Multiple-Motive Expert Systems.* Proceedings of the Fifth Technical Conference of the British Computer Society Specialist Group on Expert Systems, Warwick, United Kingdom, 1985.

[44] Stephen F. Smith, Mark S. Fox and Peng Si Ow. *Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems.* AI Magazine, 7(4):45-61, 1986.

[45] Richard Washington and Barbara Hayes-Roth. *Input Data Management in Real-Time AI Systems.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, 1989.

[46] Eric Wefald and Stuart Russell. *Estimating the Value of Computation: The Case of Real-Time Search.* Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Limited Rationality, Stanford, California, 1989.

Annex: Example in an Office Environment

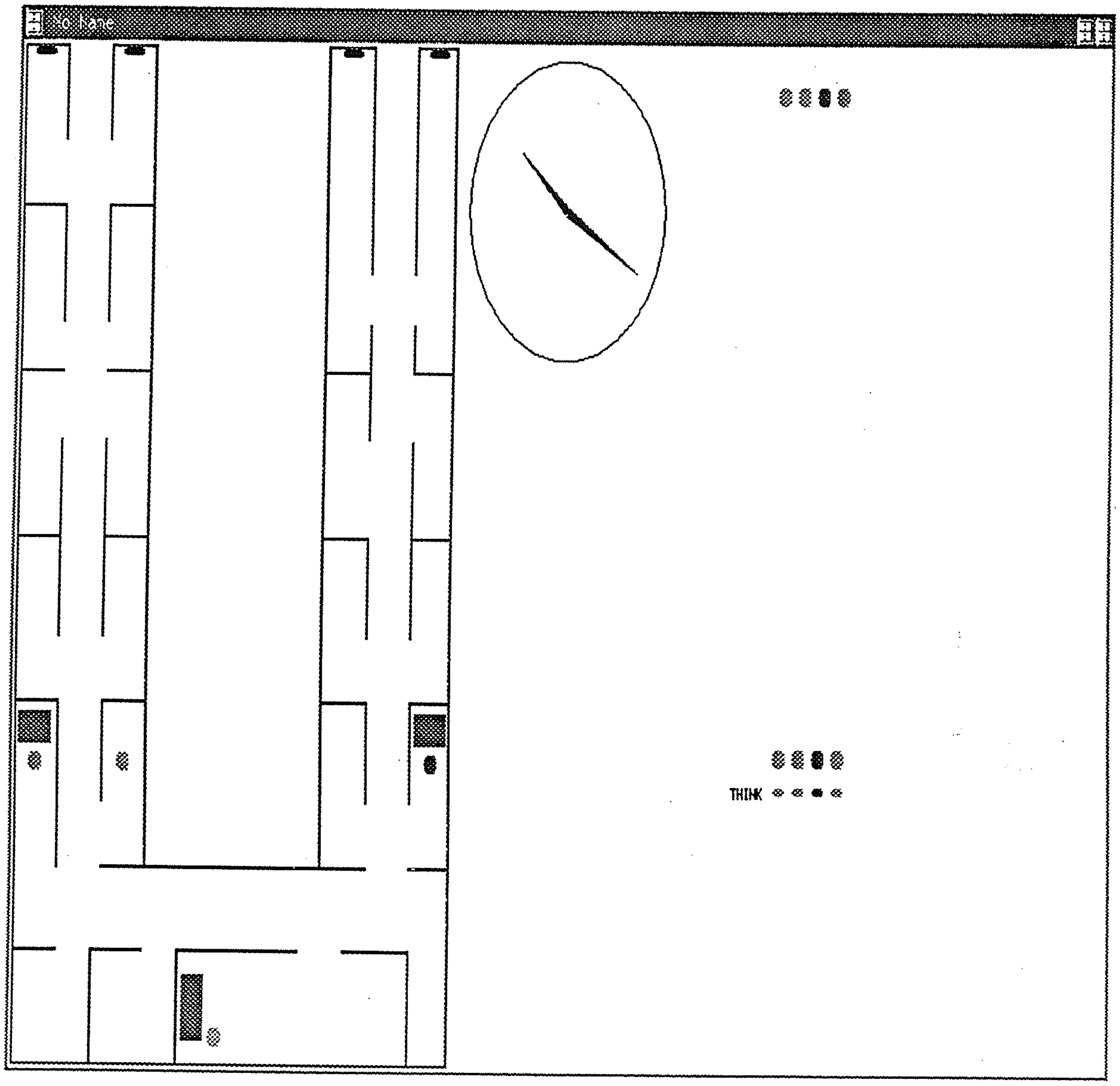
In this annex, we present a simple example in the context of the office automation project. There are four robots in the environment. These robots communicate with a central computer through infra-red communication ports. Clients order copies of books and reports. To respond to an order, a robot must get an original from one of the two square bookcases (most books and reports are available from either bookcase), go to the rectangular copier, get a copy from the copier, bring the copy to the client and bring the original back to the right bookcase. Each screen is organized as follows:

- On the left side is shown the location of each robot and object of interest in the environment.
- In the upper right corner is shown the set of proposals made by each robot for the pending orders. There is no pending order in the initial situation.
- In the lower right corner is shown the set of *starting* and *inprocess* actions performed by each robot. In the initial situation, each robot is "thinking": making plans to connect onto the communication network and see whether something needs to be done.

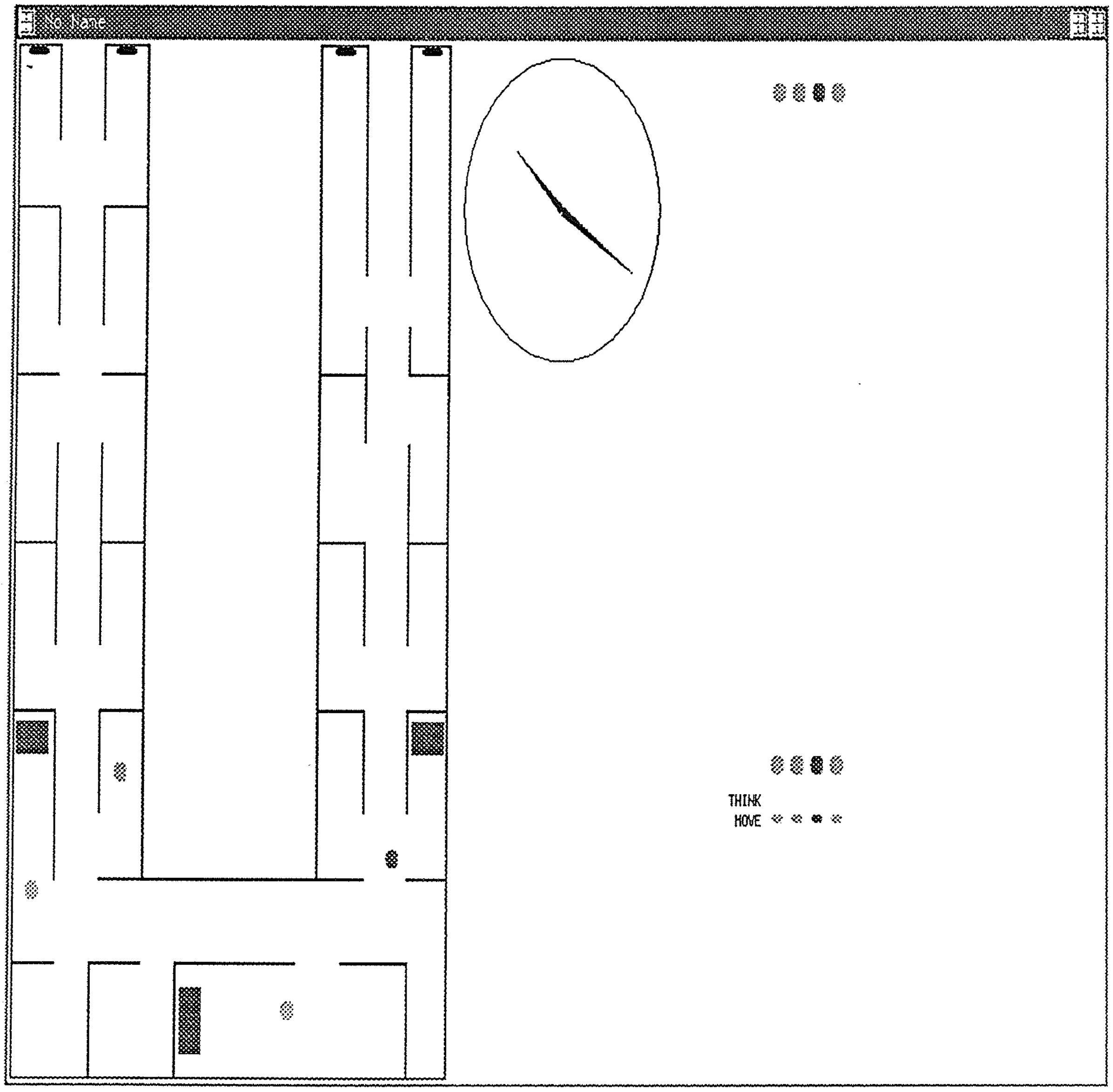
Fifteen screens are shown. Many things happen between two screens. Most of the thinking actions are skipped to keep the example small.

Screen 1

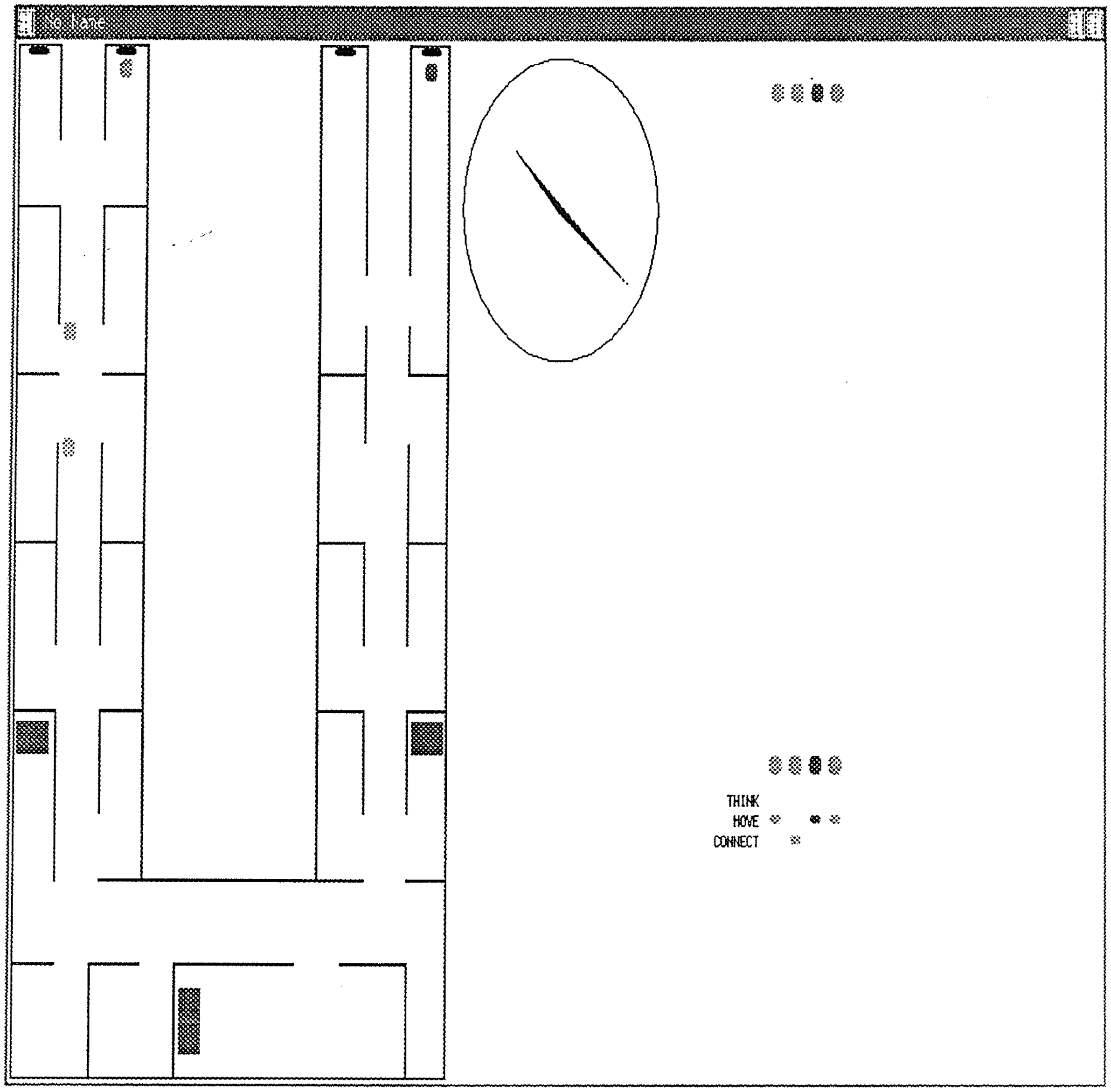
This is the initial situation. Each robot makes plans to connect onto the communication network.



Screen 2
Each robot moves toward a communication port.

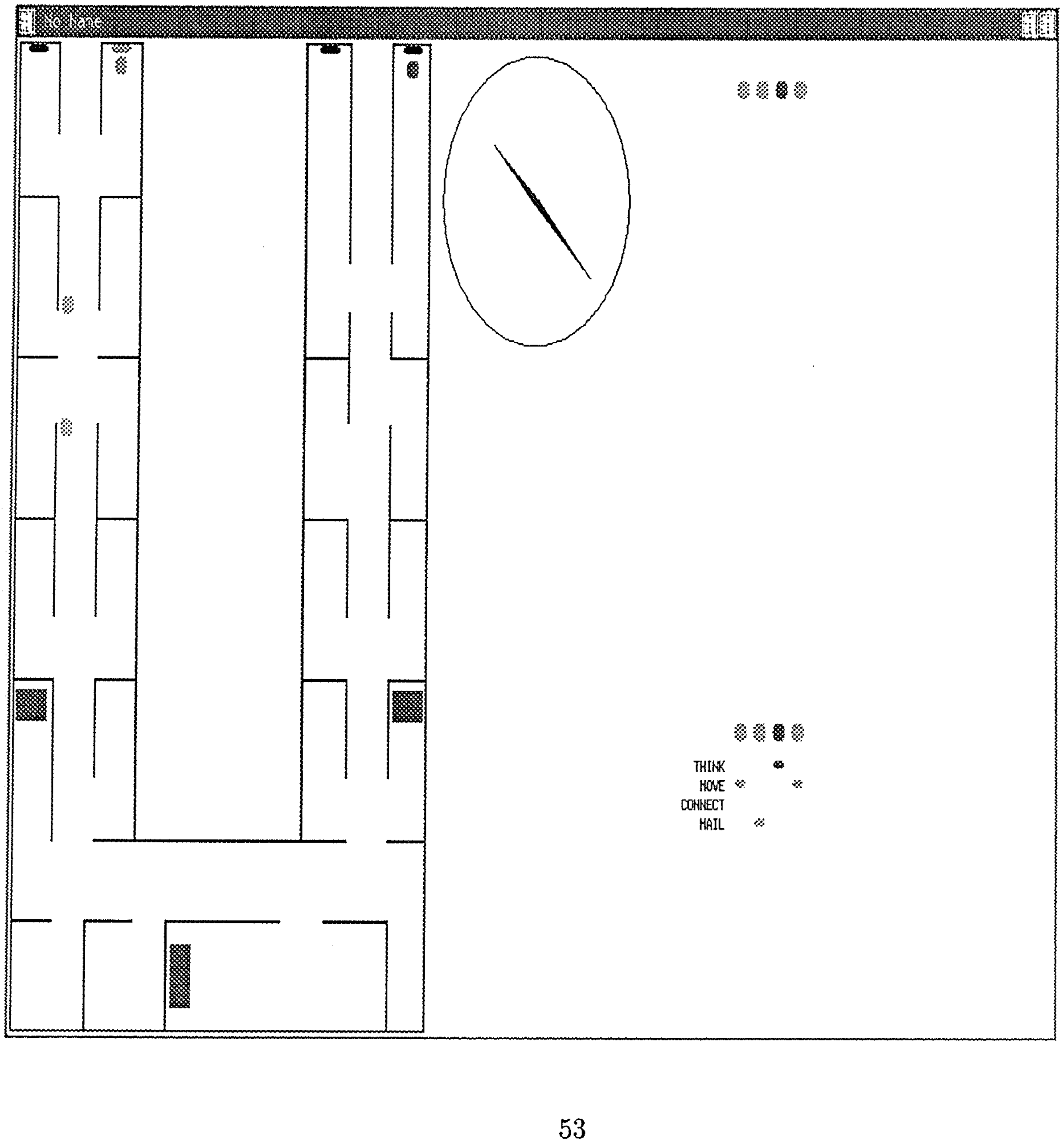


Screen 3
The second robot connects onto the communication network.



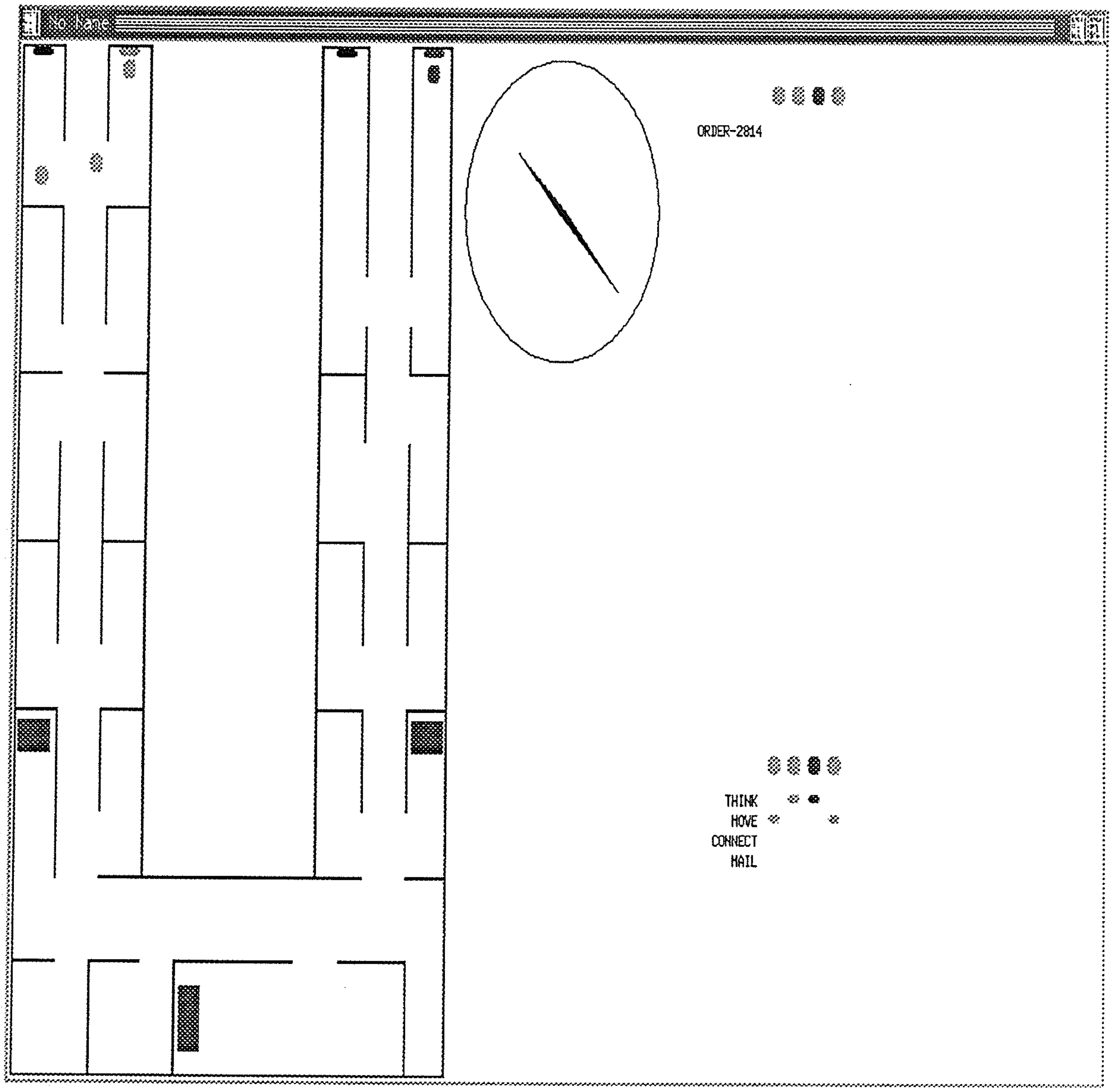
Screen 4

The second robot sends a mail to the central system to determine whether there are pending orders. The "move" action of the third robot is completed and the third robot is in the process of deciding what to do next.

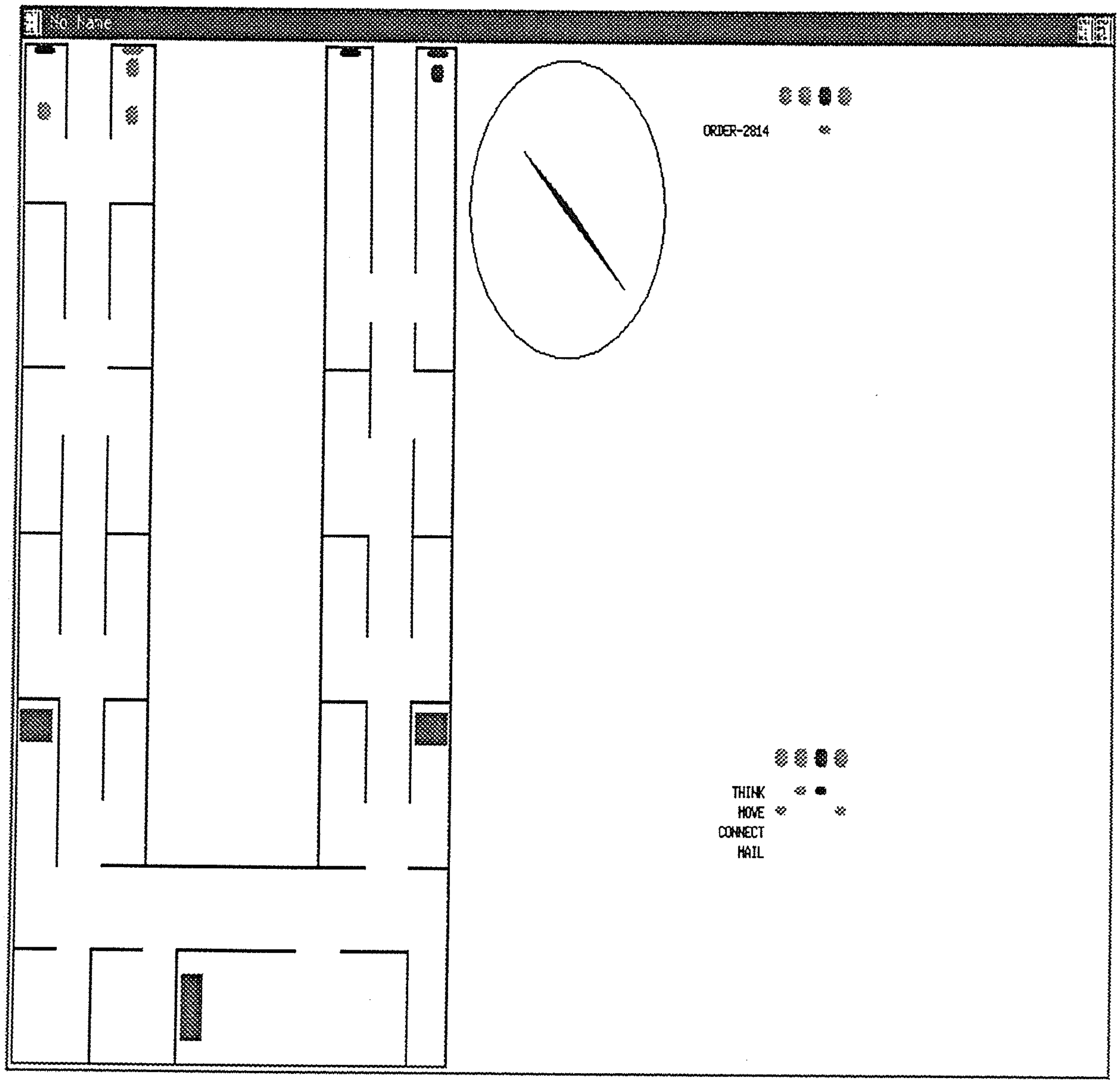


Screen 5

There is a new order. The second robot and the third robot (which is now connected) construct proposals for this order.

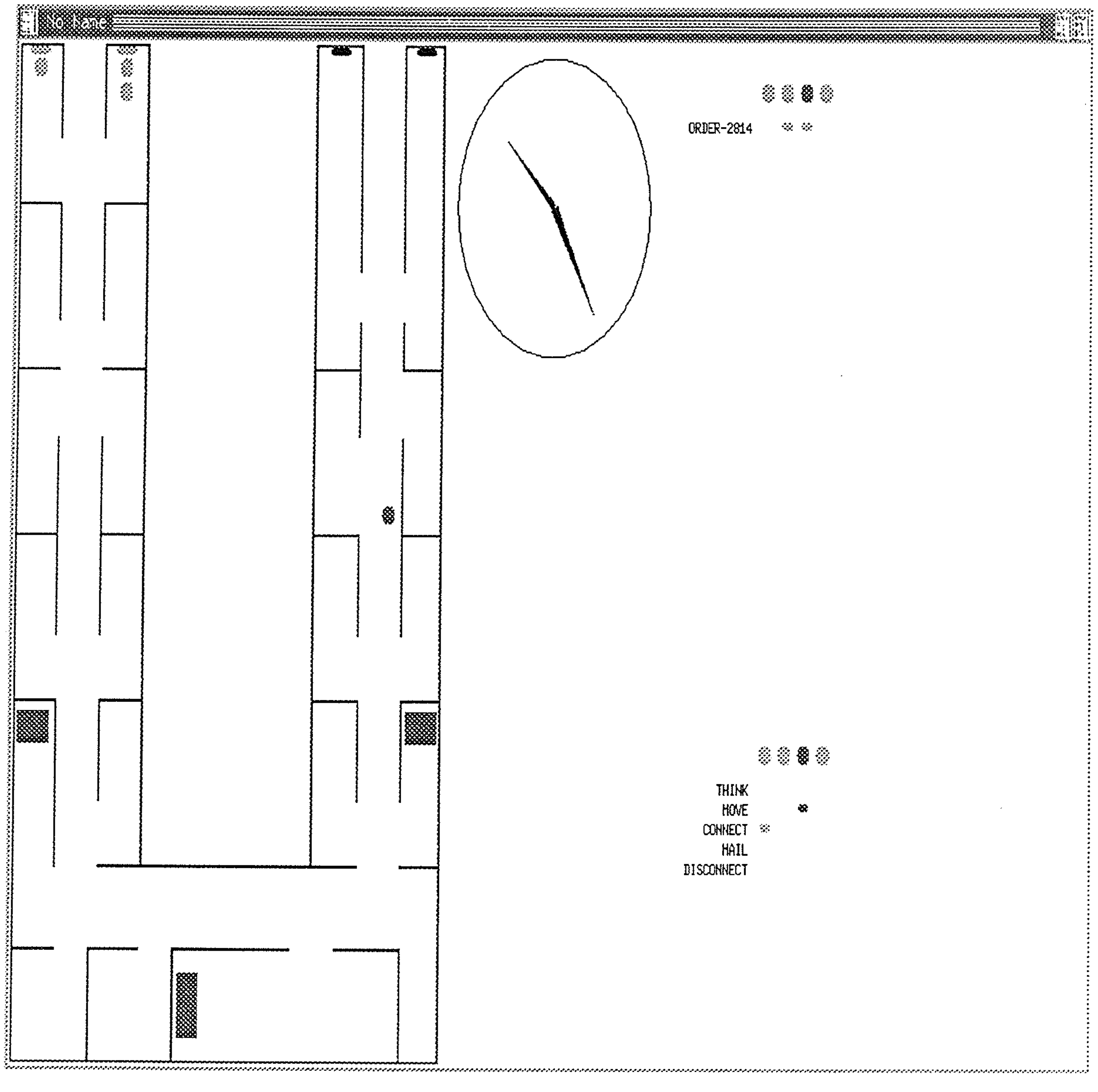


Screen 6
The proposal of the third robot is submitted to the central system.

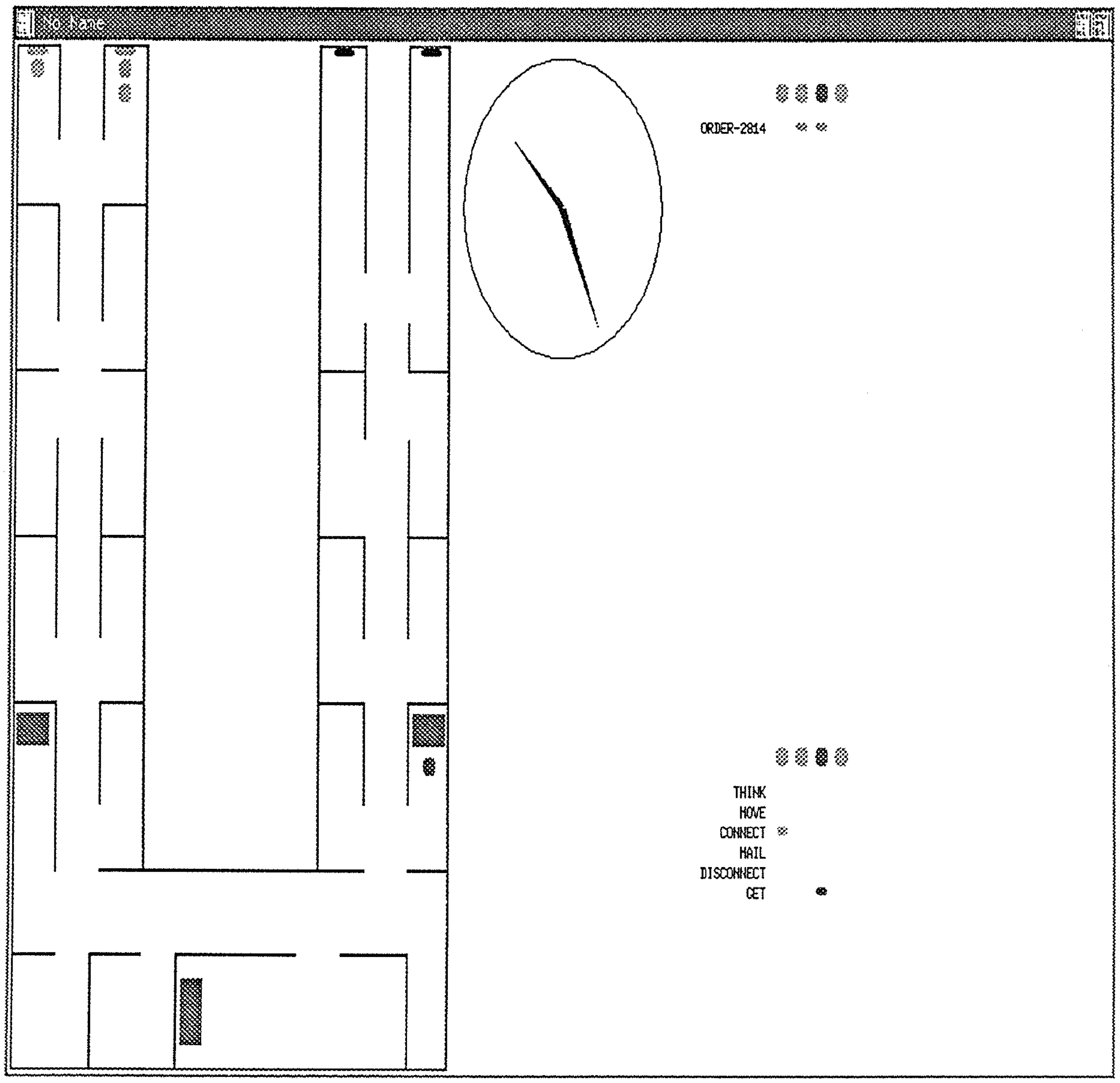


Screen 7

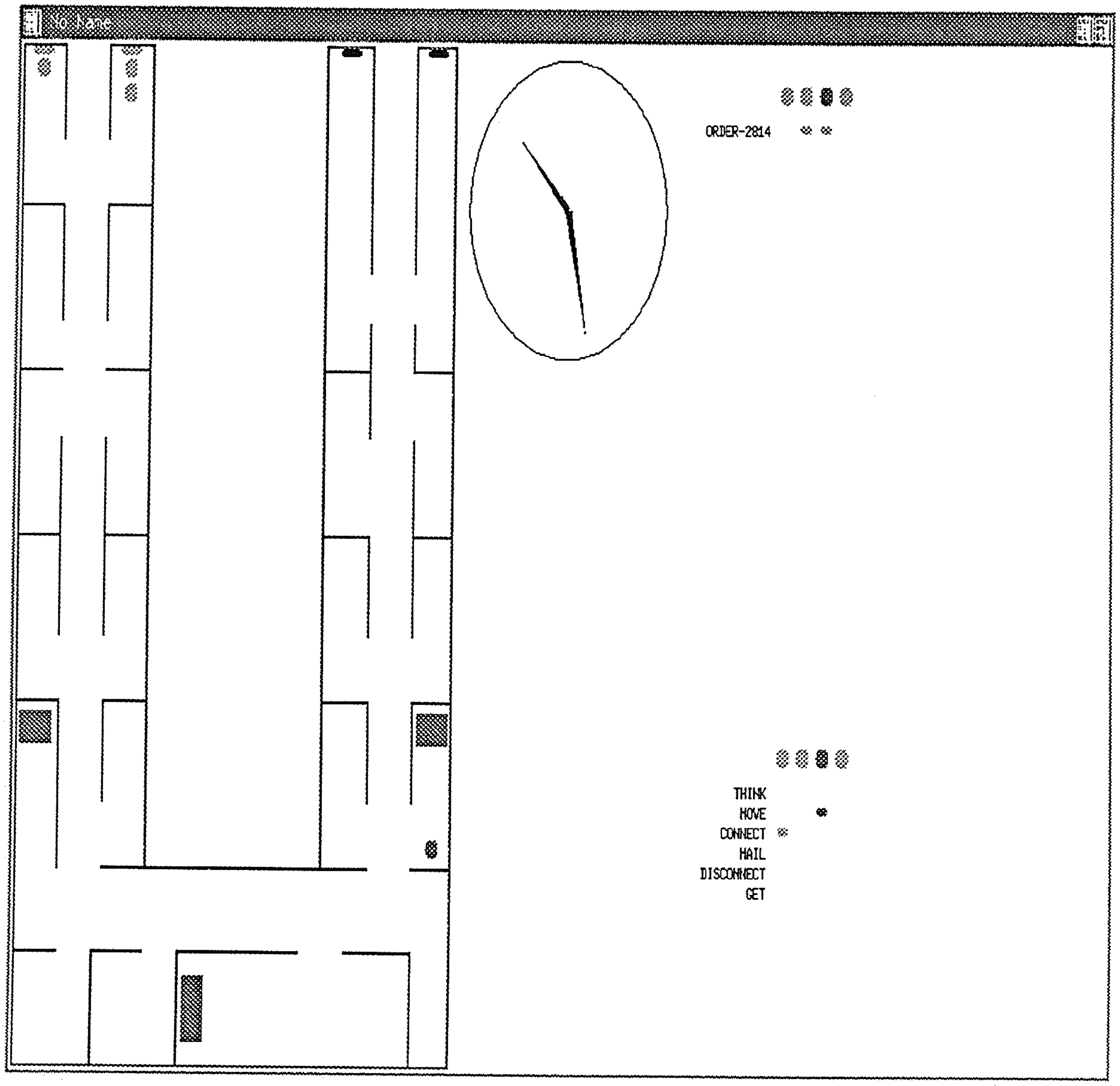
The proposal of the second robot is submitted to the central system and rejected. The proposal of the third robot is accepted and the robot is now moving. The fourth robot is now connected — but too late to make a proposal. The first robot cannot connect because the second robot is using the port: the first robot will wait.



Screen 8
The third robot gets an original from the closest bookcase.

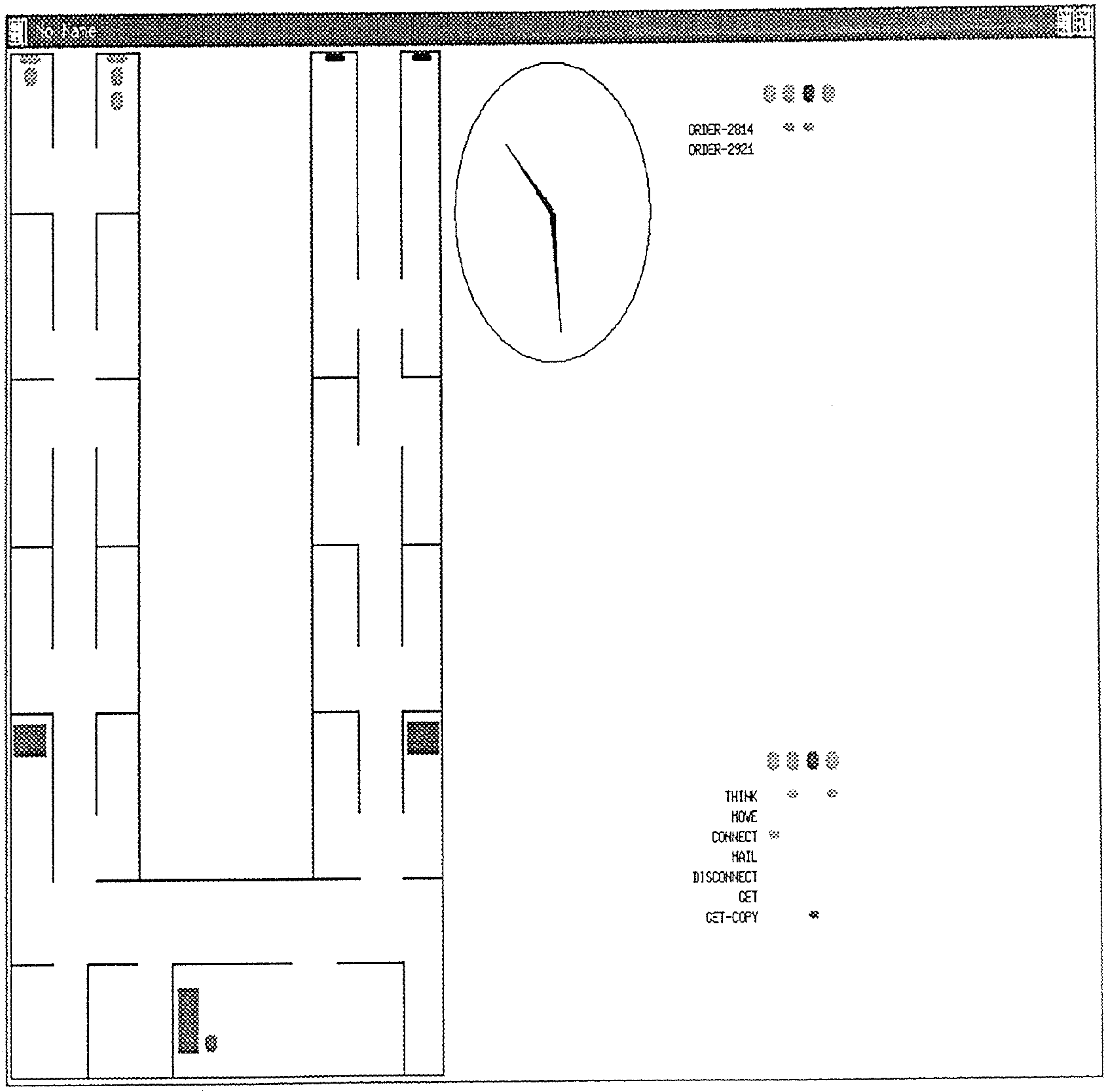


Screen 9
The third robot moves to the copier.



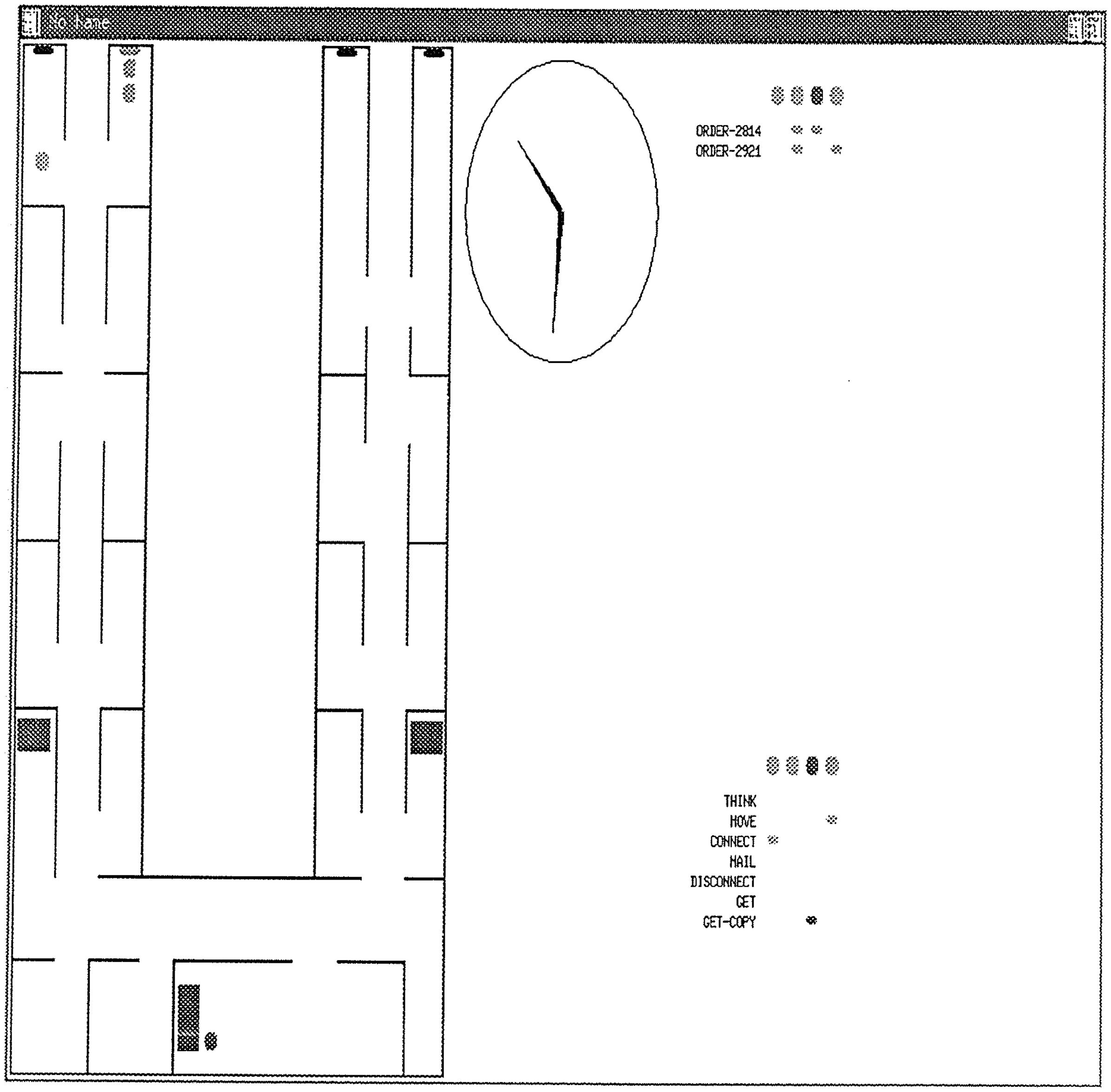
Screen 10

The second and the fourth robots construct proposals for a new order. The third robot waits at the copier (the copier is making the requested copy).

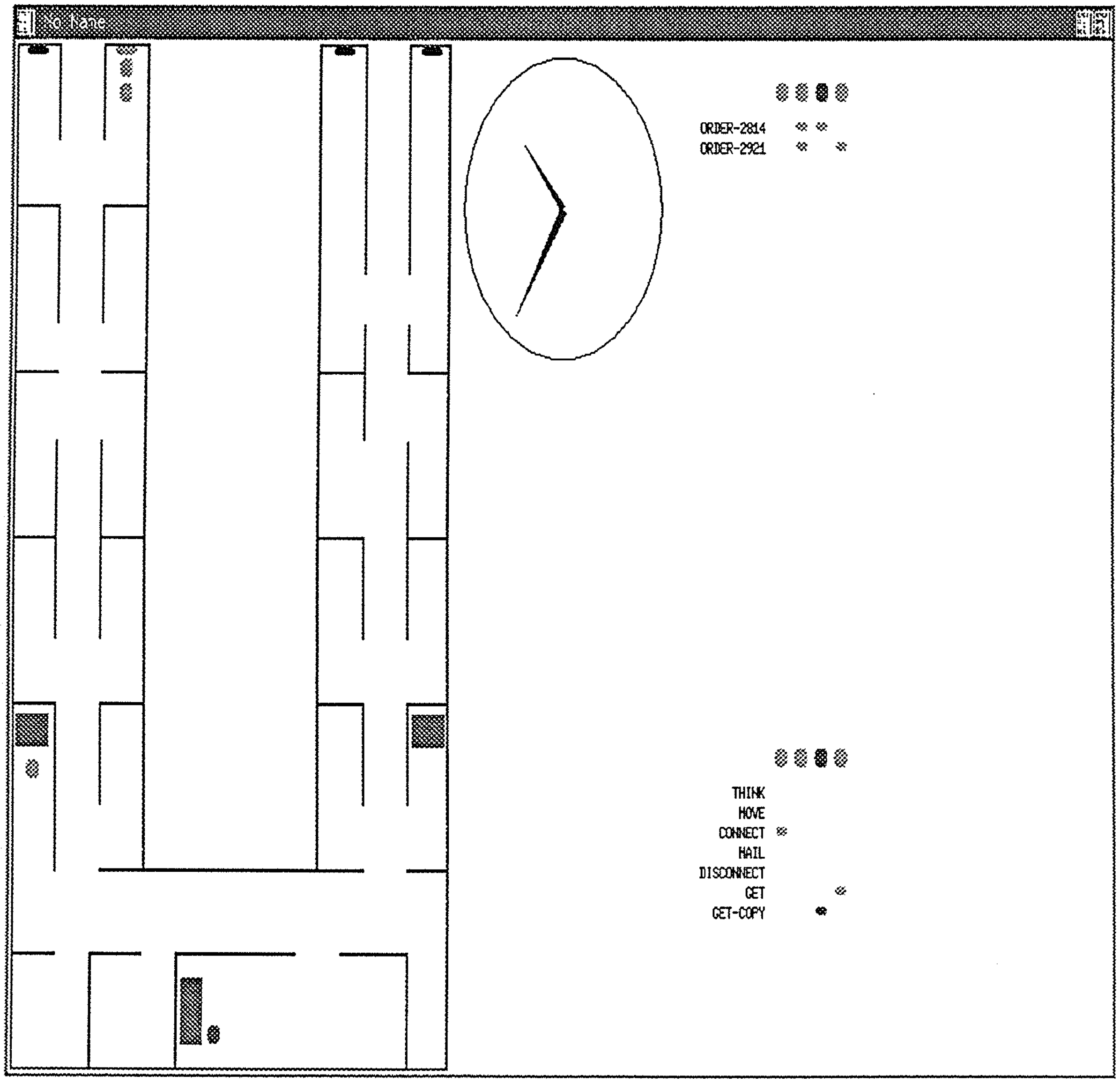


Screen 11

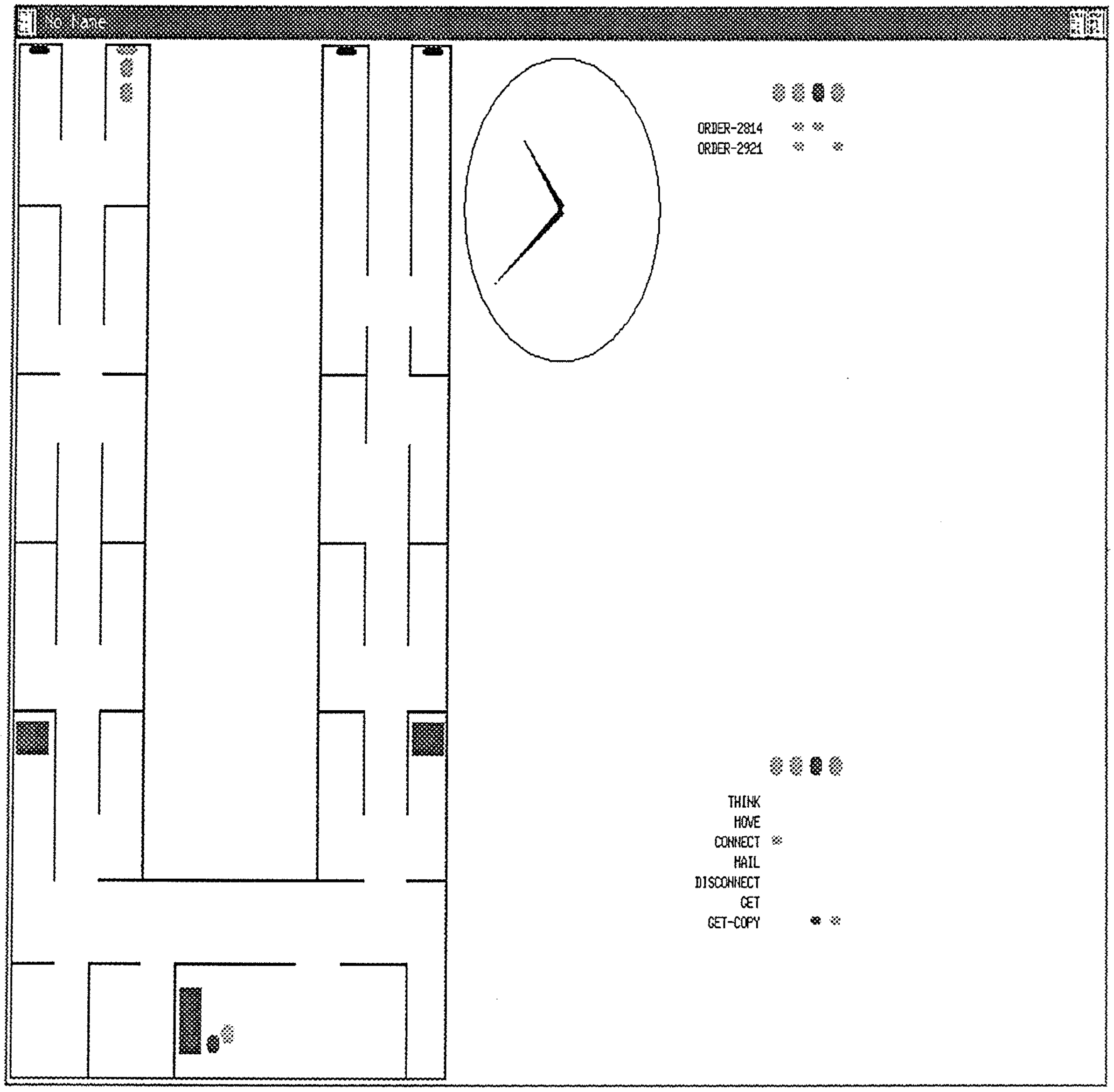
The proposal of the second robot is rejected. The proposal of the fourth robot is accepted and the robot is now moving.



Screen 12
The fourth robot gets an original from the closest bookcase.

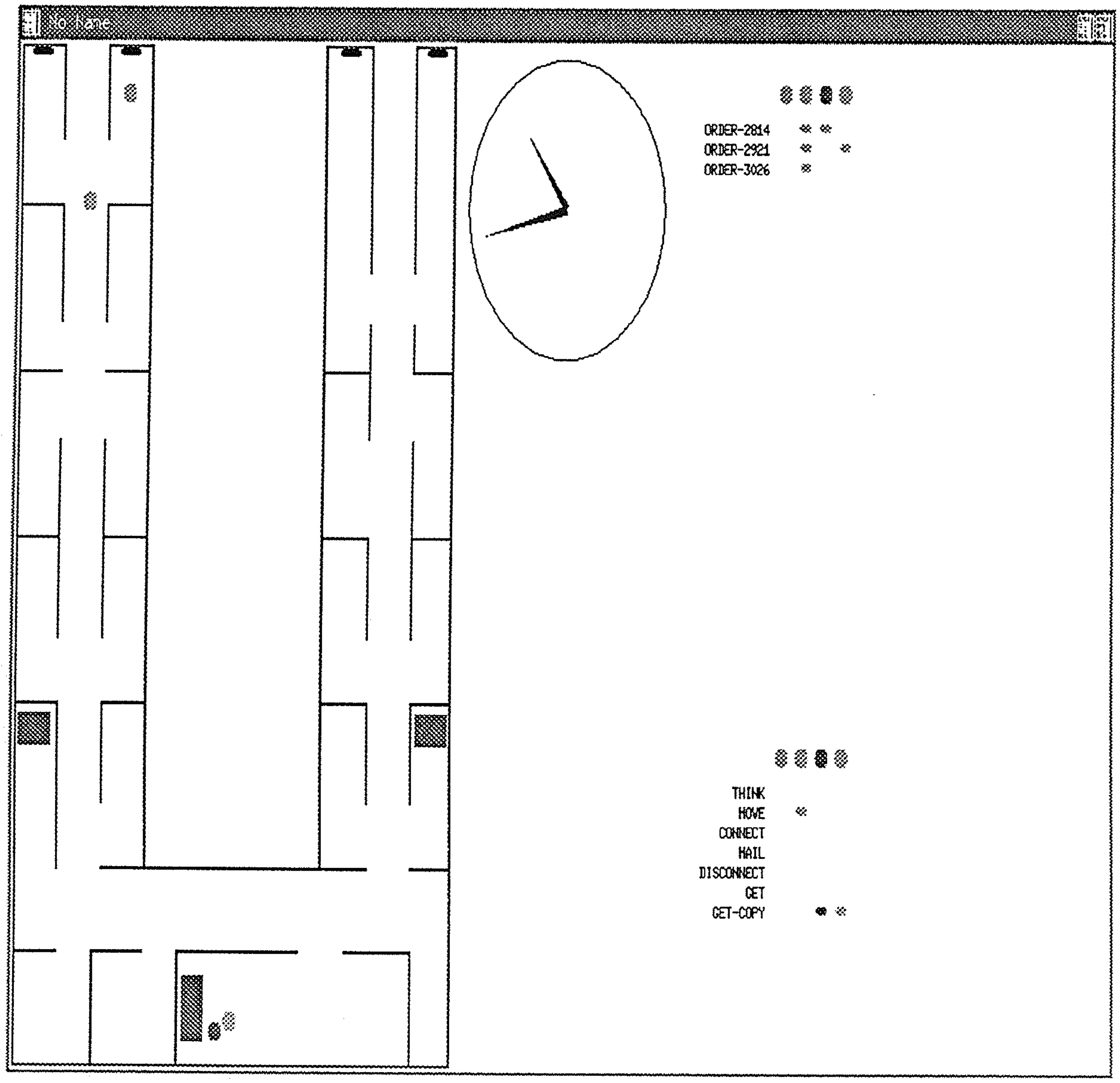


Screen 13
Two robots are now waiting at the copier.



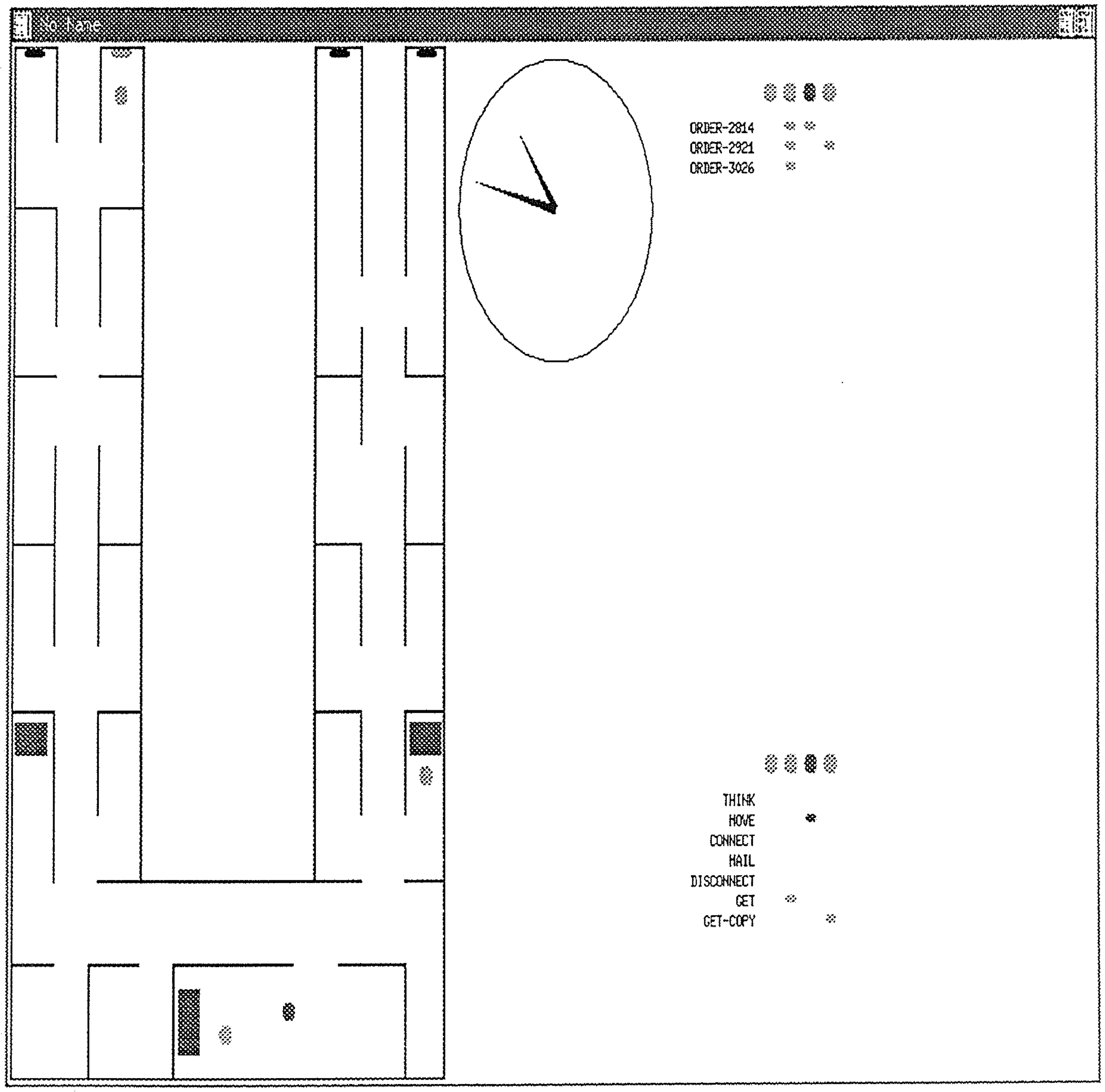
Screen 14

The second robot made a proposal for a new order. The proposal was accepted and the robot is now moving. The first robot is finally connected.



Screen 15

The second robot gets an original, available only in the farthest bookcase. The third robot got the requested copy and is now moving.



... etc.