Distributed, Parallel Simulation of Multiple, Deliberative Agents

A.M.Uhrmacher K.Gugler
Department of Computer Science
Universität Ulm
D-89069 Ulm
Germany
e-mail: lin@informatik.uni-ulm.de, kgugler@mathematik.uni-ulm.de

Abstract

Multi-agent systems comprise multiple, deliberative agents embedded in and recreating patterns of interactions. Each agent's execution consumes considerable storage and calculation capacities. For testing multi-agent systems, distributed parallel simulation techniques are required that take the dynamic pattern of composition and interaction of multi-agent systems into account. Analyzing the behavior of agents in virtual, dynamic environments necessitates relating the simulation time to the actual execution time of agents. Since the execution time of deliberative components can hardly be foretold, conservative techniques based on lookahead are not applicable. On the other hand, optimistic techniques become very expensive if mobile agents and the creation and deletion of model components are affected by a rollback. The developed simulation layer of JAMES *(a Java Based Agent Modeling Environment for Simulation) implements a moderately optimistic strategy which splits simulation and external deliberation into different threads and allows simulation and deliberation to proceed concurrently by utilizing simulation events as synchronization points.*

1 Introduction

The definition of agents subsumes a multitude of different facets [17]. Agents are reactive, deliberative, or combine reactive with deliberative capabilities. They should be sufficiently flexible to adapt to changing environments and changing requirements. Reasoning strategies allow them to anticipate the consequences of possible actions and choose the most rational action. Deliberation is typically a time and space consuming operation. Hybrid agents combine deliberation with a reactive behavior pattern to allow timely reactions within a dynamic environment. Besides these "complex" internal processes, multi-agent systems exhibit structural behavior as well [14]. Agents are mobile and solve problems by creating new software components during runtime, moving between locations, and initiating or joining groups of other software components [5].

JAMES, a Java-Based Agent Modeling Environment for Simulation, [15, 16] constitutes a framework which is aimed at supporting experiments with agents under temporal and resource constraints. Its core libraries provide the means for the description of variable structure models and their distributed, parallel execution. For that purpose, JAMES reuses and combines concepts of distributed systems and parallel discrete event simulation with ideas of endomorphy, i.e. models which contain internal models about themselves and their environment [18], and variable structure models, i.e. models whose description entails the possibility to change their own structure and behavior [15, 16]. The focus of this paper will be on the parallel simulation techniques employed.

2 JAMES - A Short Sketch

The model design in JAMES resembles that of parallel DEVS (Discrete Event System Specification) [19], enriched by means to support variable structures. Time triggered automata with the ability to assess and access their own structure are the coarse frame for the description of *BDI* agents whose attitudes comprises beliefs, desires, and intentions [15], and other agent architectures and for testing single modules, e.g. planning or learning components.

As does DEVS, JAMES distinguishes between atomic and coupled models. Testing of agents means rooting mental attitudes within the state of an atomic model, embedding deliberation, reaction and filtering of options within the state transition functions, and transforming the activities of an agent into JAMES constructs. An agent might need some time to decide which action to take, this "reaction" and "deliberation" time is translated into the atomic model's

0-7695-0677-1/00 $10.00 © 2000 IEEE

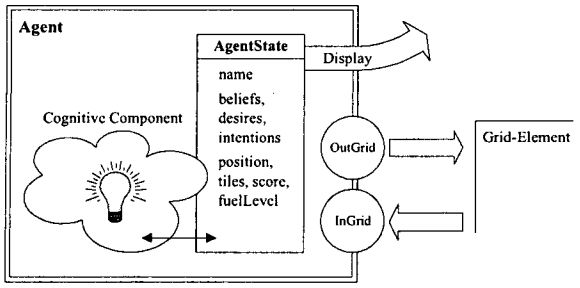

Figure 1. Agents in TILEWORLD

time-advance function. The chosen action of an agent are translated into outputs of the atomic model or into invoking JAMES methods to initiate structural changes, e.g. adding an agent to a location (Figure 1).

Atomic models are able to create new models, to add existing ones within the embedding coupled model. They can delete themselves, and determine their interaction with their environment. A model can initiate its movement from one location to another. It can initiate its move, but for its completion, i.e. for being embedded within the new context, it needs the cooperation of an on-site model. To initiate structural changes elsewhere agents have to turn to communication and negotiation in JAMES. Thus, a movement from one coupled model to another implies that another atomic model complies with the request to add the moving model into the new interaction context. To facilitate modeling, all atomic models are equipped with default methods that allow them to react to requests, e.g. to add models, to create new ones, or to delete themselves. However, these default reactions can be suppressed to decide deliberately what requests shall be executed. The range of structural changes a model can directly execute is restricted to avoid conflicts between concurrent structural changes [16].

Typically, the objective of testing is to analyze how agents cope with knowledge constraints, i.e. incomplete and uncertain knowledge, and with temporal constraints in dynamic and open environments. To test the timeliness of an agent's activities within a virtual world, it becomes necessary to relate the simulation time to the time needed for deliberation.

Some test beds define the simulated time depending on the used deliberation component and the size of the knowledge base [4], others weigh and count instructions which are executed during "deliberation". This presupposes that the deliberation component can be executed in a timed environment, e.g. "Timed Lisp" which simply overloads the standard lisp operators [1]. Most test beds determine the deliberation time as a function of the actually used computation time [3]. Even though in this case noise is induced due to changing work loads, which hampers repeatable test runs, it is the most flexible and simplistic approach in testing agent architectures and components. Neither does it presuppose a white box implementation nor a specific implementation language. Therefore, we choose this time model to facilitate the testing of different planning systems [10].

3 The Problem

Most test beds for multi-agent systems do not execute their models concurrently. They simply maintain the illusion of simultaneity on a single machine. If only a single deliberative agent is tested in a dynamic environment [1] or the coordination strategies of a moderate number of reactive agents [7], there is no need for a distributed, parallel execution of agents. To efficiently test more than a single deliberative agent, which consumes significant space and computation resources, a concurrent, distributed simulation layer is necessary.

As a first step, we adopted the DEVS parallel simulator in JAMES to exploit the parallelism inherent in the model. As does DEVS, Parallel DEVS [2] associates with each atomic model and each coupled model a simulator and a coordinator, respectively. Thus, the compositional model finds its pendant at the execution level in a hierarchy of *processors*, i.e. *simulators* and *coordinators*, controlled by a so-called *root coordinator*. In this hierarchy, coordinators are associated with coupled models, and simulators with atomic models. The latter form the leafs of the processor tree.

As does Parallel DEVS, JAMES propagates messages, so called *-messages, top down the model tree each time an event is scheduled. Each processor knows how many inputs its associated model will receive at the current time step. It waits for these inputs, which are forwarded by the #-message, to arrive and executes the transition function. Messages indicating the completion of a transition, so called done-messages, report the component's time of next event, and, optionally, structural changes to the coordinator. Each coordinator waits for all its activated components to finish their transition, executes the indicated structural changes, and sends a summary to its own coordinator. Eventually, a done-message reaches the root coordinator. Thereafter, the next simulation step is initiated by the root coordinator.

The problem with this approach is apparent. The time base of JAMES is "quasi continuous" and only events which happen exactly at the same simulation time are processed concurrently. The virtual time an agent needs for generating a plan is determined based on the time it needs for computation. Thus, the event of two or more agents deliberating concurrently will be extremely rare. Not surprisingly, distributing the execution has degraded the performance of most parallel test runs.

The agents' transition functions call the deliberation sys-

tem, e.g. the planning system. Depending on the time actually needed by the planning system, each agent will determine its time of next event. The time of next event forms a necessary part of the done-message which is sent by the simulator to the coordinator. The problem is that each transition - and thus the simulator and the entire simulation - waits for the planning system to complete. Only thereafter, the time of next event can be determined, and only then a done-message can be sent to the coordinator. Since each coordinator waits for all its components to complete, i.e. for their done-messages, before it sends a done-message to its own coordinator, the entire simulation is blocked. The absurdness is that the simulator waits for the time actually needed for generating the plan to announce its time of next event to its coordinator, rather than for the results. The results will not be needed until the simulation has advanced to the time of next event, which might be in the far future. However, all other events are blocked until the time of next event, the termination of the deliberation, is scheduled, which unfortunately necessitates executing the deliberation.

If the simulator knew this time in advance a planning system could run concurrently with the rest of the simulation until the time at which the results are needed by the simulation, i.e. the time determined as the completion time of the plan. However, this time cannot be predicted. The question is whether it can be guaranteed that the plan generation will take at least a certain amount of time to complete. Unfortunately, as Logan and Theodoropolous summarize their experiences [12], lookaheads are very difficult to determine for deliberative agents. The execution time of deliberative components varies within one and the same scenario drastically - as not only experiences of AI researchers show in general, but our experiences with testing planning systems in particular. We tested two agents equipped with the planning system GraphPlan in the TILEWORLD scenario. In our first experiments, the generation of a plan needed between 2 seconds and 20 hours on a ULTRA 2 [15].

Due to the variance of the deliberation time a conservative strategy based on lookaheads seems questionable. An alternative are optimistic techniques. Optimistic schemes do not strictly avoid causality errors but detect and recover from them. A so called *straggler event* sent by its influencers indicates that a component is ahead of its influencers in time. In response the component has to roll back to the state before the straggler event happened. It annihilates outputs sent, if any, with time stamps later than that of the straggler event and proceeds by re-processing all the input events from the time of the straggler event.

Keeping track of prior states might lead to a storage problem - in JAMES even more so, since not only the state but also the structure of the overall model has to be recorded. Agents move, delete themselves, change their couplings and add new components. These structural events are subject to rollbacks as any other value changes within a component. Thus, it is crucial to determine a time horizon prior to which information can be discarded [6] and to keep this time horizon close to the current simulation time.

4 Toward a Solution

The basic idea is not to wait until the planner is completed but to create a separate external thread and to return a message which indicates that a deliberation process is under way. Thus, the transition function and the overall simulation can proceed. However, as in other synchronization protocols, e.g. Moving Time Windows [11] and Bounded Time Warp [13], barrier synchronizations are introduced to prevent the simulation from proceeding too far ahead compared to the external processes still running. To prevent cascading rollbacks over several simulation steps the simulation ensures at each step that it is safe to proceed.

At each step the simulator does not only activate the models with imminent events but also those still deliberating. It applies the *real-time-knob* function of the model to the time consumed so far by the deliberation process. This function relates deliberation time to simulation time. If the "thinking" consumed a sufficiently large amount of time to make a completion prior to the current time impossible, the simulator proceeds. Otherwise it waits until it is either safe to proceed or the deliberation is finished. Thus, there is no need to roll back farther than to the last event and a rollback will only require the storage of one state.

For integrating real-time processes into the simulation, the definition of models in JAMES is extended: z describes a port which is filled by an external source and accessed read-only by the model's functions. The $real-time-knob$ relates simulation and deliberation time. Models are equipped with methods that allow starting external programs as separate threads. A transition function invokes an external program by using these methods. In any case, transition and initialization functions will finish without waiting for the results of the external program. Thus, the simulation can continue.

The simulation system shall allow external, internal, and confluent events to take place while an external program is active. If an agent is represented as an atomic model (and not as a coupled model with different specialized components), this enables an agent to react to external events while it is planning or learning.

4.1 Simulator

The simulator of a model is activated by the *-message, which indicates an internal, external, or confluent event.

when an input $(*, xCount, t)$ has been received

```
am is the associated model
am_old = ∅
inpCount = xCount
outCount = 1
busy_fixed = false
if t = t_finished charge z
if t_start ≠ ∞
  block until
      t_start + real-time-knob(used-time) > t
      ∨¬busy
  busy_fixed = busy
  if ¬busy_fixed then
    (* planner finished just now *)
    t_finished = t_start + real − time − knob(used − time)
    finished_backup = t_finished
    t_start = ∞
    if t_finished ≤ t then rollback = true
  endif
endif
t_min = min(t_next, t_finished)
if ¬rollback ∧ (t = t_min ∨ (t < t_min ∧ xCount > 0))
  s_old = s
  t_old = t_last
  if t = t_min then
    send (λ(z, s)) to parent
    if xCount = 0 then
      s = δ_int(z, s)
    else
      block until inpCount = 0
    s = δ_con(z, s, xb)
  endif
else
  block until inpCount = 0
  s = δ_ext(z, s, t − t_last, xb)
endif
if ¬busy_fixed ∧ busy then
  t_start = t
  busy_fixed = busy
endif
am_old = am
am = ρ(s)
if t = t_finished then
  t_finished = ∞
  flush(z)
t_last = t
t_next = t_last + ta(s)
endif
send (done, min(t_next, t_finished), varStrucRequest(s),
      outCount, busy_fixed, rollback)
      to parent coordinator
end
```

If at the current time the completion of a deliberation process is scheduled, the port z is charged with the results of the deliberation process. A value of t_{start} less than infinity indicates that an external process is running (parallel to the rest of the simulation). In this case, the simulator blocks until the time used for deliberating has reached the current simulation time or until the deliberation is finished. Since the *busy* flag can be set asynchronously by the deliberation system any time, $busy_{fixed}$ is introduced to ensure a consistent execution. If the deliberation has been finished, the times of last and next event are determined. If the deliberation finishes before the time of the *-message the completion marks a straggler event and the variable `rollback` is set to true to initiate a rollback. If the variable `rollback` is true the simulator will return a done-message to the coordinator which indicates that a rollback is necessary.

If the deliberation component is still running or no deliberation component is running to begin with the *-handler proceeds as usual. It applies the appropriate state transition function, followed by the model transition function ρ which determines whether a new model structure shall replace the old one. To support a rollback the old state, the old time of last event and the old model structure are recorded. If no internal, external, or confluent event is due and no rollback occurred the simulator has only been checked and it returns its old time of next event with the flag $busy_{fixed}$ still set to true. Whether or not an external thread has been started during a transition function is checked by $\neg busy_{fixed} \wedge busy$. The thread controlling the external process must not reset the busy flag if it is finished before this expression has been evaluated. Otherwise the simulator will not notice that a planner had been started. The simulator holds two slots for memorizing the "normal" time of next event (t_{next}) and the completion time of the planner ($t_{finished}$).

The #-handler is responsible for collecting inputs. It remains unchanged by the revised procedure. It collects the inputs and increases the semaphore which will finally kick the *-handler into action.

```
when an input (#, y, t) has been received
  block until inpCount > 0
  xb = xb + y
  inpCount = inpCount − 1
end
```

A new handler is introduced, the `rollback`-handler. If the `rollback`-handler receives the request to perform a rollback from its coordinator it checks whether a rollback was requested by itself. Otherwise, it has to update its associated model's state. If a rollback refers to a time at which the agent started deliberating $t_{start} = t_{last}$ the external thread has to be stopped and the time t_{start} is set to infinity.

If a structural change is affected by a rollback, the old model structure has to be reinstalled (e.g. transition, output, and time advance functions). Afterwards the rollback handler resets the values of the old state.

```
when an input (rollback) has been received
  if rollback then
    rollback = false
  else
    if t_start = t_last then
      stopdeliberation :
      t_start = ∞
      busy_fixed = false
    endif
    if am_old¬am ∧ am_old¬∅
      am = am_old
      am_old = ∅
    endif
    if t_last = finished_backup then
      t_finished = finished_backup
    t_last = t_old
    am = am_old
    s = s_old
    t_next = t_last + ta(s)
    t_min = min(t_next, t_finished)
  endif
  send (done, t_min, outCount, busy_fixed, rollback) to parent
end
```

4.2 Coordinator

We also need a `rollback`-handler at the level of the coordinator - not only to inform its components but also to roll back if a structural change has been executed. But let us first inspect the necessary changes within the *-handler.

```
when an input (*, xCount, t) has been received
  varStruc = ∅
  n_old = ∅
  n is the associated network
  IMM = {d ∈ D | t_next_d = t}
  OM = {d | d ∈ IMM ∧ outCount > 0}
  BM = {d ∈ D | busy_d}
  INF = {d ∈ D | ∃i ∈ OM.i ∈ I_d ∨ (xCount > 0 ∧ d_N ∈ I_d)}
  for each r ∈ INF ∪ IMM ∪ BM
    iCount_r = Σ_{d∈I_r∩OM} outCount_d
    if d_N ∈ I_r then
      iCount_r = iCount_r + xCount
    send (*, iCount_r, t) to r's processor
    actCount := actCount + 1
  end
  block until actCount = 0
  t_old = t_last
  t_last = t
  rollback = ⋁_{d∈D} rollback_d
  busy = ⋁_{d∈D} busy_d
  if ¬rollback then
    if varStruc¬∅ then
      n_old = n
      n = ρ(varStruc2Do)
    endif
  endif
  t_next = minimum{t_next_d}
  outCount = Σ_{{d∈D|d∈I_{d_N} ∧ t_next_d = t_next}} outCount_d
  send (done, t_next, ∅, outCount, busy, rollback)
    to parent
end
```

When a coordinator is activated by a star message no structural changes are pending and no storage of an older version of the network is necessary. In PDEVS, processors are only activated to indicate an internal, external, or confluent event of their associated model. Imminent and influenced components respectively their processors ($IMM \cup INF$) have to be informed. In this version, processors that have a deliberation process running (BM) are activated as well.

Based on the number of imminents which produce outputs (OM) and the existing coupling, the coordinator *-handler calculates the number of inputs each component will receive. Afterwards the coordinator waits for all its activated components to send a done-message. If no rollback has been indicated by any of its components it processes the required structural changes. The old network is stored in case somewhere else in the overall model a causal error will be detected. Delaying the execution of structural changes until all transitions have been completed avoids conflicts between concurrent structural and non-structural changes.

If, within this coupled model, no rollback is necessary the time of next event and the number of outputs to be produced at the next internal or confluent event are determined. Together with the rollback and busy flag, this information is sent to the parent coordinator which proceeds likewise.

```
when (done, t_next, varStrucRequest, outC, b, r)
        has been received from processor p
  block until actCount > 0
  actCount = actCount − 1
  varStruc2Do = varStruc2Do ∪ varStrucRequest
  update information about the sender:
  rollback_p = r
  busy_p = b
  t_next_p = t_next
  outCount_p = outC
end
```

When a done-message is received the done-handler decreases the number of done-messages to be received and updates the information about the receiver (rollback and busy flag). Finally, the time of next event and outputs to be produced are recorded. It adds the received request to change the structure of the network to its $varStruc2Do$. The coordinator executes the requested structural changes only if no rollback is announced within this coordinator.

```
when an input (#, y, t) has been received
  forward outputs (x_i, t) produced by i
      according to Z_{i,j}
    if j = d_N then to parent coordinator
    else to component j
end
```

The #-handler at the coordinator level remains unmodified.

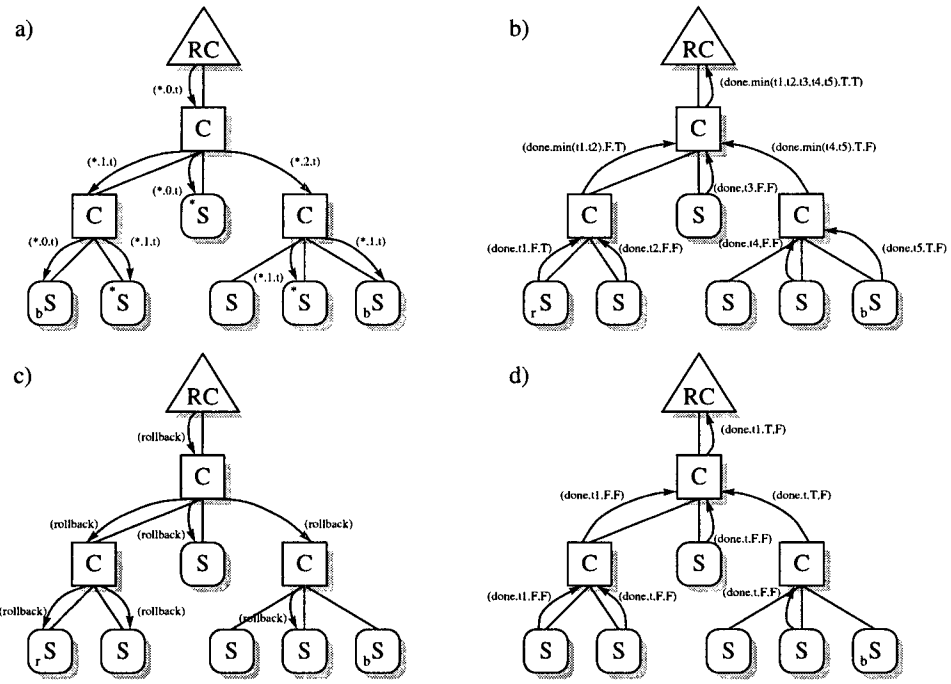

Figure 2. Message passing during rollback: **a)** An event is announced by propagating *-messages top down. Processors marked with (*) have to process an internal, external, or confluent event while processors marked (b) have a planner running. **b)** The planner of the bottom left model has just finished planning and a rollback becomes necessary. The root-coordinator receives this information through the done messages. **c)** All models that processed an event at t receive a rollback message and restore their old state. **d)** The time of next event is determined by propagating done messages.

```
when an input (rollback) has been received
  if ¬rollback then
    n = n_old
    n_old = ∅
  endif
  for each r ∈ IMM ∪ INF ∪ {d ∈ BM|rollback_d = true}
    send(rollback) to r's processor
      actCount = actCount + 1
    endif
  end
  block until actCount = 0
  t_next = minimum{t_next_d}
  outCount = Σ_{d∈D|d∈I_dN ∧ t_next_d = t_next} outCount_d
  busy = ⋁_{d∈D} busy_d
  send (done, t_next, ∅, outCount, busy, false)
end
```

If a rollback reaches a coupled model it checks whether it is already aware of a rollback. In this case no structural changes have been executed. Otherwise the structural changes executed at the network level have to be undone by installing the old state of the network, i.e. its old components and the couplings which existed among them. Afterwards, the components are informed about the rollback.

After the components completed the rollback operation, the rollback-handler of the coordinator will determine the time of next event, the number of outputs to be produced at the next time step and the busy flag and send its done-message to its own parent coordinator. The rollback flag can be set to false since no successive rollbacks can occur.

4.3 Root Coordinator

The root coordinator controls the simulation by sending *-messages indicating the time of the next event in the abstract simulator. This triggers the processing of events in the processor tree which is eventually confirmed by a done-message from the topmost coordinator.

```
t_next = t_next(topmost coordinator)
repeat until t_next > t_EndOfSimulation ∨ (t_next = ∞ ∧ ¬busy)
  if rollback
    send (rollback) to topmost coordinator
  else
    if busy ∧ t_next = ∞ then
      t_next := estimate_t_next()
    send (*, 0, t_next) to topmost coordinator
```

```
wait for (done,t,outCount,b,r) from topmost coord.
t_next := t    busy := b    rollback := r
```

If a causality error has been detected the root coordinator initiates the execution of the rollback through a `rollback`-message. Otherwise the simulation proceeds as usual. There is another problem the root coordinator has to handle: if there are deliberation processes running and no events are scheduled, i.e. $t_{next} = \infty$ has been returned to the root coordinator. In this case, the root coordinator sends a *-message with an estimated t_{next}, typically a number close to infinity. This will cause the simulator to wait until at least one of the deliberation processes has finished and another event (completion of the process) can be scheduled.

5 Evaluation: Agents in TILEWORLD

Initially, TILEWORLD was developed to test different control, particularly commitment, strategies of IRMA agents [9, 8].

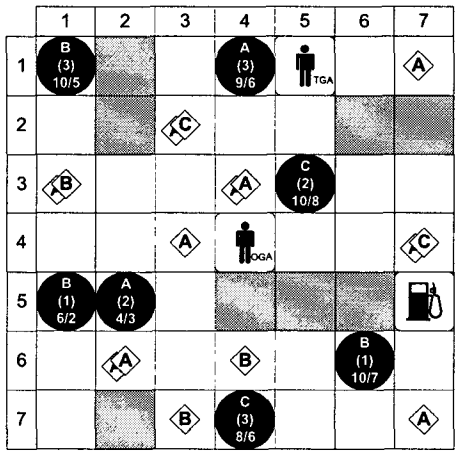

Figure 3. A TILEWORLD scenario [10]

TILEWORLD (Figure 3) is a two dimensional grid world with tiles, which can be moved, and holes, which should be filled with tiles. There are obstacles, which impede the movement of agents, and gas stations which allow the refilling of consumed energy. Tiles, holes, and obstacles appear and disappear at certain rates, according to global parameter settings. Thus, the environment displays probabilistic, dynamic behavior.

The effectiveness of an agent is measured in terms of scores that summarize the number and kind of holes filled, and the type of tiles used for filling. TILEWORLD combines a counting problem, how many more tiles of what type does the agent need to fill a particular hole, with route planning in a grid world. This setting puts only few constraints on the search space and implies a costly deliberation with respect to computing time and memory.

The TILEWORLD scenario we have chosen comprises an 8 by 8 grid, 1000 units of simulation time, and a real-time knob, i.e. factor, of 1. Thus, 1 unit of simulation time should be about 1 second. The grid elements change, e.g. holes and tiles appear and disappear, every 50 time units with a probability of 40%. All agents we tested had a scan range of 5 grid elements, limited, but sufficient fuel, and were planning for two goals simultaneously. Within our implementation of TILEWORLD scanning requires intensive message exchange. The experiments were run on 2 Ultra 2 machines equipped with about 200 MB each. Each experiment consisted of 15 runs.

We first put our algorithm to test using one single agent in the TILEWORLD. The time the simulation runs needed to complete averaged slightly less than 1200 seconds. About 150 of those were due to the scanning activity, and more than 900 were due to planning.

Afterwards, we added another agent to the scenario. For the experiment JAMES distributed the model and the processor tree. The agents and their simulators, including their planners, were running on different machines. Each of the agents was planning an average of more than 900 seconds. The total time used for the simulation averaged about 1450 seconds. About 250 of those were due to the scanning activity of the agents.

Thus, the 900 additional seconds of planning time for the second agent caused almost no additional overhead in simulation time. The overhead caused by the sending of rollback messages turned out to be negligible. However, we did not measure the effort required for saving the state of the model. Not surprisingly, running two agents on a single machine, requires about twice the computation time. The scanning is of course faster but this does not compensate for the loss of efficiency caused by the sequential execution of the planners.

6 Conclusion

The testing of multiple, deliberative agents is space- and time consuming. External modules are plugged into a frame provided by the test bed. The frame provides the interface between agent and agent-architecture to be tested and the virtual environment agents shall be tested in.

Since the performance of agents depends significantly on their timely decisions, a time model is employed to relate the actual or expected execution time of agents to the virtual time of the test environment. One time model is often applied due to its flexibility and simplicity: it clocks the execution of the deliberation component and applies a function to transform the consumed time into simulation time. Thus, only after the generation of a plan the simulation will

know at what time to schedule the completion of a deliberation process.

The proposed approach splits simulation and external deliberation into different threads. We allow simulation and deliberation to proceed concurrently by utilizing simulation events as synchronization points. The simulation is delayed to guarantee at each step that no rollback beyond the last state can occur. The simulation proceeds only if the time used by the deliberation process exceeds the current time step in simulation time. Thus, stepwise, the entire simulation and the deliberation processes approach the wallclock and simulation time at which a deliberation component will finally complete its execution. On the way, other agents can start and finish deliberation, models that constitute the environment of an agent can proceed with their dynamics. Looking at performance, one can say that our algorithm simulates several planning agents close to the cost of a single agent, given that a sufficient number of machines are available.

References

[1] S.D. Anderson. Simulation of Multiple Time-Pressured Agents. In *Proc. of the Wintersimulation Conference, WSC'97*, Atlanta, 1997.

[2] A.C. Chow. Parallel DEVS: A Parallel Hierarchical, Modular Modeling Formalism. *SCS - Transactions on Computer Simulation*, 13(2):55–67, 1996.

[3] P. R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. *AI Magazine*, 10(3):32–48, 1989.

[4] E. H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers, Boston, 1988.

[5] M. R. Genesereth and S. P. Ketchpel. Software Agents. *Communications of the ACM*, 37(7):48–53, 1994.

[6] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *SCS Distributed Simulation Conference*, pages 63–69, 1985.

[7] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The SWARM Simulation System: A Toolkit for Building Multi-Agent Simulations. http://www.santafe.edu/projects/swarm, June 1996.

[8] M. E. Pollack, D. Joslin, A. Nunes, U. Sigalit, and E. Eithan. Experimental Investigation of An Agent Commitment Strategy. Technical Report 94-31, University of Pittsburg, Department of Computer Science, 1994.

[9] M. E. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *AAAI-90*, pages 183–189, Boston, MA, 1990.

[10] B. Schattenberg. Agentenmodellierung und -evaluierung im Rahmen eines objekt-orientierten, verteilten Simulationssystems. Master's thesis, University of Ulm, Department of Computer Science, 1998.

[11] L. Sokol, D. Briscoe, and A. Wieland. Mtw: A strategy for scheduling discrete simulation events for concurrent execution. In *Proc. of the SCS Western MultiConference on Advances in parallel and Distributed Simulation.*, pages 169–173, 1988.

[12] G. Theodoropoulos and B. Logan. A Framework for the Distributed Simulation of Agent-Based Systems. In H. Szczerbicka, editor, *European Simulation Multi Conference - ESM'99*, pages 58–65. SCS Europe, Ghent, 1999.

[13] S. Turner and M. Xu. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proc. of the 6th Workshop on Parallel and Distributed Simulation*, pages 117–126, 1992.

[14] A.M. Uhrmacher. Concepts of Object- and Agent-Oriented Simulation. *Transactions of the Society of Computer Simulation*, 14(2):59–67, 1997.

[15] A.M. Uhrmacher and B. Schattenberg. Agents in Discrete Event Simulation. In *European Simulation Symposium - ESS'98*, Nottingham, October 1998. SCS.

[16] A.M. Uhrmacher, P. Tyschler, and D. Tyschler. Modeling and Simulation of Mobile Agents. *Future Generation Computer Systems*, (to appear 2000).

[17] M.J. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[18] B. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models - Intelligent Agents and Endomorphic Systems*. Academic Press, San Diego, 1990.

[19] B.P. Zeigler, H. Praehofer, and Kim T.G. *Theory of Modeling and Simulation*. Academic Press, 1999.