UNIVERSITÀ DELLA CALABRIA

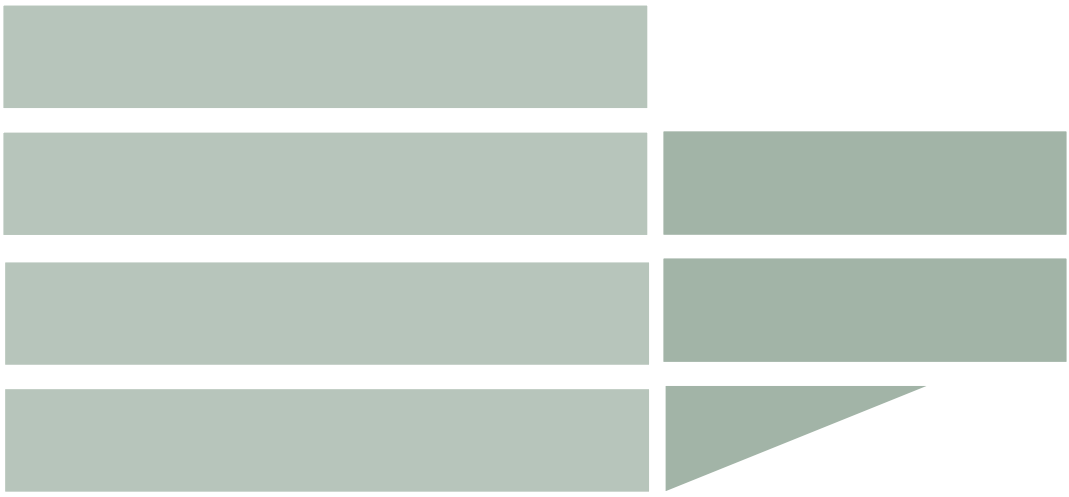
Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXIII ciclo

Tesi di Dottorato

Agent architectures for modelling and parallel simulation of complex systems

Andrea Giordano



UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXIII ciclo

Tesi di Dottorato

Agent architectures for modelling and parallel simulation of complex systems

Andrea Giordano

Coordinatore
Prof.
Luigi Palopoli

Supervisore
Prof.
Libero Nigro

DEIS

DEIS- DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Novembre

Settore Scientifico Disciplinare: ING-INF/05

In memory of my father

Acknowledgements

I would like to thank prof. Libero Nigro for his professionalism and dedication with which he supervised my research work.

I acknowledge all my colleagues at L.I.S., and especially Ing. Franco Cicirelli for its nearness and friendship during my PHD activity.

I would like to thank my family and my friends who have always sustained me in the important situations of my life.

Finally, I thank also the group 'precari invisibili della ricerca' which keeps the fight alive with passion and intelligence.

Preface

A fundamental step in systems engineering lifecycle is represented by requirements specification and thorough property analysis before proceeding with design down to the implementation phase. What are needed are suitable modelling languages able to ensure the necessary rigor in the expression of system behaviour. Property analysis can hopefully be based on exhaustive verification (e.g. by model checking) but more often has to resort to techniques such as *discrete-event simulation* (DES) which although is unable to reproduce the full execution state graph of a modelled system, can anyway provide valuable insights in the functional and behavioural properties of a system.

Modelling and analysis get further complicated when dealing with large-scale systems, like telecommunication systems, biological or social or economic systems, artificial intelligence systems, weather forecasting systems and so forth, which normally can only be studied through simulation. Such systems challenge for the avalability of adequate modelling languages, runtime executive infrastructures and powerful computing architectures for high-performance execution.

Nowadays, *agent-based computing* is often regarded as a very promising paradigm for *modelling and simulation* (M&S) of complex systems. Agents are typically encapsulated software entities which can naturally be associated to system components, and interact and coordinate to one another towards the achievement of a common goal (system mission or emerging behavior from agent individual behaviors). Agents favour software modularity and reusability and can allow an exploitation of e.g. a *distributed execution* context. Although exist several agent-based modelling languages and tools, this work argues that solutions are often unsatisfactory to provide the flexibility and the execution performance required by complex systems M&S. This PHD thesis develops some M&S solutions based on a particular agent-based framework in Java -Theatre- proposed in the Software Engineering Laboratory (www.lis.deis.unical.it), headed by Prof. L. Nigro, of the Department of Informatics Electronics and Systems Science of the University of Calabria. Theatre is a light-weight architecture centred on the *actor* computational model. It

characterizes by its ability to support a huge number of agents in a simulation model, and to work with different transport layers and standard or emerging middleware like *HLA* or *Terracotta*. Theatre can effectively be used for M&S of large and scalable systems, and can act as an efficient runtime system for supporting several formal modelling languages. Specific contributions of this thesis are in the following areas:

- customizing Theatre for supporting hierarchical actors and *statecharts*, thus controlling the explosion of states in situations where even at the agent level the dynamic behavior becomes complex
- experimenting with Theatre for supporting *DEVS* particularly in the case of simulator interoperability by web services
- using Theatre for distributing *RePast* models for high-performance simulation of complex agent-based systems over HLA/RTI
- developing a Theatre kernel on top of Terracotta to enable parallel/distributed simulation of large systems on a multi-core cluster
- evaluating the use of Theatre in supporting *spatial environments* organizations in situated multi-agent systems.

The thesis is structured in 6 chapters as follows.

- *Chapter 1* is devoted to a presentation of fundamental concepts of agent-based computing.
- *Chapter 2* focuses on some exemplar M&S formalisms, HLA and Terracotta middleware for parallel/distributed simulation, and to well-known agent-based toolkits like RePast. The chapter covers also basics of aspect oriented programming.
- *Chapter 3* presents the Theatre actor-based architecture and reports some experience in M&S of variable structure systems. In addition details of a mapping of Theatre on HLA are discussed and applied in the context of a conservative time management algorithm.
- *Chapter 4* describes two particular developments: (a) statecharts based actors and (b) a DEVS implementation in terms of Theatre.
- *Chapter 5* illustrates an actor kernel based on Terracotta and reports about its performance using a multi-core cluster.
- *Chapter 6* presents some techniques for handling spatial environment partitioning of situated multi-agent systems. The chapter also focuses on a software engineering project and its concretization, concerning the distribution of RePast agent models using the Theatre architecture over HLA.

Rende, November 2010 *Andrea Giordano*

Contents

1

Concepts of Agent-based Computing

1.1 Introduction

Agent-based computing is a promising approach for the development of applications in complex and distributed scenarios where no global control is possible. It provides a way to design and develop software applications in terms of autonomous entities, referred to as *agents*, which are situated in an environment and flexibly can achieve their goals by interacting to one another by means of high-level protocols and languages [Jen01, OZ04]. Compared to other computing paradigms, agent-based computing can be defined as *evolutionary*, however in terms of practical usage it appears to be *revolutionary* [Lev04].

Currently, agents are the focus of intense interest in many sub-fields of computer science and artificial intelligence. Agents are being used in an increasingly and incredible wide variety of applications [LMP04, JW98], ranging from comparatively small systems such as email filters to large, open, complex, mission-critical systems such as air traffic control. Knowledge Management, Team Working, Bio-Inspired Architecture, Supply Chain Management, Entertainment and Health Care are growing areas for agent research and development, especially when complex interactions among entities in the same environment make it extremely difficult to understand and thoroughly analyse system behaviour.

An agent is a high-level software abstraction which is a key for modern software because: (i) the autonomy of application components reflects the intrinsically decentralised nature of distributed systems and can be considered as the natural evolution of notions like modularity and encapsulation; (ii) the flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable scenarios where software is expected to operate.

Although in widespread use, agent computing cannot be considered as a ‟panacea″ in computer science: nothing that can be done with agents cannot be done with other means [HCK97, WJ98]. However, research areas involved and the number of deployed applications related to agents bear testament

of potential advantages [Jen01, HCK97, Pic01, Lev04, GKCR00, BHR+02, Bra97] of this approach.

The chapter is structured as follows. Section 1.2 provides basic definitions and concepts related to the agent metaphor. A discussion about agent architectures is provided in section 1.3. 1.4 offer a survey on some methodologies, technologies and infrastructures relevant to the agent paradigm. Finally, conclusions are presented with an indication of current directions in agent research.

1.2 Agent metaphor

The term *agent* literally means *one that acts* or *one that does something* [Woo02] and it origins from the dog latin *"to do"*. A tacit assumption is that *agents* take an active role and originate actions that can affect their environment, rather than passively allowing their environment to affect them.

Even if the above definition is very simple and clear, there is no general consensus on what *software agents* really are. Probably this is due to the sheer breadth of subject areas in which agents are being applied (see section 1.1).

The concept of *agent* can be traced back to in the 1970s to Carl Hewitts concurrent Actor model [Hew77a]. In this model, Hewitt proposed a self-contained, interactive and concurrently executing object which he termed *actor*. This object has some encapsulated internal state and can respond to messages coming from other similar objects.

Above statement provides an *operational* definition of an agent but it owns a low level of abstraction in describing agent behaviour. Franklin and Graesser [FG97] critically reviewed a number of agent definitions showing that none of them is perfect or complete. Bradshaw [Bra97] instead characterises agents in terms of *ascriptions* and *descriptions*. A classical definition of agent is owed to Wooldrige and Jennings [WJ95]:

> *...the term agent is used to denote a hardware or (more usually) software-based computer system that enjoys the following properties:*
>
> - ***autonomy**: agents operate without direct intervention of humans or others, and have some kind of control over their actions and internal state*
> - ***social ability**: agents interact with other agents (and possibly humans) via some kind of agent-communication language*
> - ***reactivity**: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined) and respond in a timely fashion to changes that occur in the environment.*

This provide the so called *weak definition of an agent*. However for some researchers the term agent has a stronger and more specific meaning than that

sketched out above, so in a *strong definition*, an agent can hold also some of the following properties:

- ***mobility****: the ability of an agent to move across a network*
- ***veracity****: the assumption that an agent will not knowingly communicate false information*
- ***benevolence****: the assumption that agents do not have conflicting goals, and that every agent will therefore always try to do what is asked to it*
- ***rationality****: (crudely) the assumption that an agent will act in order to achieve its goals, and will not act in such a way to prevent its goals being achieved at least insofar as its beliefs permit.*

An agent (see figure 1.1) can be viewed as a problem-solving entity with well-defined boundaries and interfaces [Jen01] that can act on behalf of an user, another agent or itself. Agents are situated (or better embedded) in an environment upon which they have *partial control* and *partial observability*. An agent senses the environment through sensors and acts on it through actuators. *Autonomy* means that agents can choose how and if respond to some external stimuli. In other words an agent *can say no* if it wants. *Reactiveness* and *proactivity* mean instead that an agent is able to respond in a timely fashion to external changes that occur in it's environment and take initiatives.

An environment provides the conditions under which an agent can exist and it defines the properties of the world (e.g., laws, rules, constraints and policies) in which an agent will function. Agents may also use the environment as communication channel to exchange messages to one another (see section **??**). It's important to describe precisely such an environment since a slight change in it can impact the results of the agent system in an unpredictable way. Environment attributes include accessibility, determinism, diversity, controllability, volatility. Each of them heavily affects agent behaviour and capabilities. Currently there are no standardised way to describe these important features, and to clearly differentiate them from the agent code [Lev04].

In adopting an agent-oriented view, it soon becomes clear that modelling or dealing with most problems requires or involves more than one single agent. The issue is to choose many *simpler* and *smaller* agents or a reduced

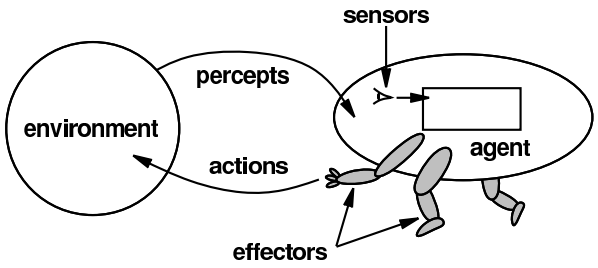


Fig. 1.1. An agent representation

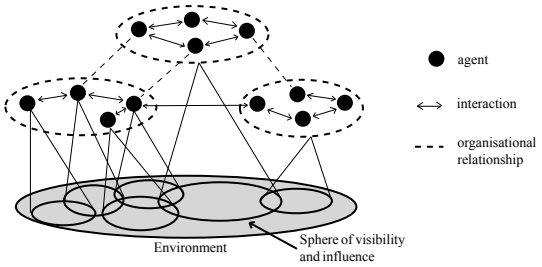


Fig. 1.2. A canonical view of a MAS

number of *more complex* agents. It's impossible to give an ultimate answer to this question. It depends on, for instances, to the intrinsic nature of the problem to solve, to the degree of modularity and the abstraction level, the kind and number of relations occurring among system entities, the kind of properties that are required to study or to emphasize, the expertise of the system developer. A system in which multiple agents are involved is usually referred to as a Multi Agent System (MAS):

...[an agent] is in a multi-agent system (MAS) that contains an environment, objects and agents (the agents being the only ones to act), relations among all the entities, a set of operations that can be performed by the entities and the changes of the universe in time and due to these actions [Fer99].

A MAS can be viewed as a loosely coupled network of problem solvers (i.e., agents) that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver which in turn has only a limited viewpoint of the problems themself. In a such system there is no global control, data are decentralised and computation is intrinsically asynchronous. Improving system reliability, availability and efficiency are some of the advantages in using a MAS (see [Syc98]). A MAS, however, is more than a bunch of agents which merely interact. Agents can form societies in which every participant exhibits roles and owns rules (see figure 1.2).

Agent interactions can vary from simple semantic operations (for instance through traditional client/server communications) to rich social interactions (i.e., the ability to cooperate, coordinate, and negotiate about a course of actions). An agent communication language (see section **??**) permits to maintain relationships among agent acquaintances and, as in human society, these relationships are dynamic: they can evolve, change or be broken.

The notion of an *environment*, together with *social ability*, are key features which distinguish agent-based computing from other computing paradigms like Object Oriented Programming [Lev04, Woo97]. With respect to other paradigms, the *agent metaphor* provides a convenient and powerful way to properly describe *real world pictures* and to deal with scenarios in which

collective behaviour and interaction may produce *emerging properties* that cannot be achieved by single entities [NNS05, SS03, BSdS+05]. It's also effective in systems with high runtime uncertainty or incomplete information (e.g., telecommunication service across multiple providers) and when, in a dynamic environment, there are some decisions to take based on multiple sources and large amounts of data (e.g., e-markets, logistics).

Further agent disquisition should require at this point to speak about *agent classification* and *agent taxonomy*. Agents can be better qualified in term of their purposes and their behaviours like degree of mobility, degree of autonomy and so forth. These issues are beyond the scope of this chapter, the reader is referred to [FG97, Bra97, Nwa96].

1.3 Architectures

Agent architectures can be thought of as software engineering models of agents. Research in this area is concerned with the problem of designing software or hardware systems that will support various agent properties and behaviours.

Maes [Mae91] defines an agent architecture as a methodology for building agents which specifies how agents can be decomposed into a set of interacting modules whose aim is to model how *perceptions* and the *internal state* of an agent may determine an *action* and changes in the *agent state*. Agent architecture can be divided in three broad categories [WJ95]: *deliberative*, *reactive* and *hybrid*. The proposals are fundamentally different in their view of the term of *"intelligence"*.

1.3.1 Deliberative Architectures

Rely on the reduction of the world to a representation of realisable symbols that can be combined to form structures upon which processes can be executed to operate upon the symbols according to a coded set of instructions [NS76]. Decision regarding actions to perform are made via logical reasoning, pattern matching and symbolic manipulation. From a theoretical point of view two issues exist. The first one concerns translating the real world in an accurate, adequate and useful symbolic representation (*transduction problem*), the second one concerns on how symbolically represent information about real-world entities and process, and how to get agents to reason upon these information (*representation/reasoning problem*).

The idea of having a deliberative agent, i.e., able to exhibit a *"rational behaviour"*, is very attractive but, from a practical point of view, it is difficult to build useful symbol manipulation algorithms that will be guaranteed to terminate with useful results in an acceptable time bound.

A particular type of deliberative architecture, perhaps the most known example, is the *BDI architecture* [BIP88, RG95], which is founded on rational agents having certain mental attitudes expressed in term of *Belief*, *Desire* and

Intention. The BDI architecture draws its inspiration from the philosophical theories of Bratman [BIP88] who argues that *intentions* play a significant and distinct role in practical reasoning that cannot be reduced only to beliefs and desires. Beliefs, desires and intentions represent respectively the *information*, *motivational* and *deliberative* states of an agent. These *mental attitudes* determine the system behaviour and are critical for achieving adequate or optimal performance when deliberation is subject to resource bounds.

Beliefs can be viewed as the informative components of system state which has to be appropriately kept updated after each sensing action.

An agent requires also to have information about the objectives (or *desires*) to be accomplished and, more generally, what priorities and payoff are associated with them. Unlike the system beliefs, these information are usually generated instantaneously, or functionally, without requiring any state representation. This component is representative of system desires which can be thought of as representing the motivational state of the system or what an agent wishes to become true. Adopted desires are often called *goals*.

Taken actions, or changes into environment, may affect system beliefs and desires. Thus it becomes necessary to include a component into system state to represent the *currently chosen course of actions*. This additional state component represents the agent intentions which, in essence, capture the deliberative component of an agent system or what an agent will try to make true. An intension also captures the notion of *commitment* with respect to a previous decision. A commitment embodies the balance between reactivity and goal directedness of an agent-oriented system. *Static* and *dynamic constraints* [RG95] may be added to relate together beliefs, desires and intentions in order to characterise agent behaviour. These constraints, for instances, are particularly important to ensure stability when frequent changes occur into the agent environment.

A whole range of practical development efforts related to BDI systems have been undertaken. These include for instances IRMA [BIP88], COSY [BS92] and GRATE* [Jen93] which are all reviewed in [HS96]. Deepening on BDI-architecture, and related BDI-logic, can be found instead in [RG91, Rao95].

1.3.2 Reactive Architectures

They rely on the basic and fundamental assumption that *"intelligent behaviour"* may be achieved without using complex symbolic representations [Bro90, Bro91a, Bro91b]. Intelligence is assumed to be an emergent property of certain complex systems which arises as a result of interactions among agents and interactions between agents and their environment. This philosophy presumes that *real intelligence* exists only in the real world and it is not in disembodied systems such as theorem provers or expert systems.

Explicit symbolic models and symbolic reasoning mechanisms disappear within *reactive architectures* which in turn provide an opposed point of view with respect to deliberative architectures.

One of the first, and most known, reactive architecture is the *subsumption architecture* [Bro86].

A subsumption architecture is a hierarchy of possible concurrent task-accomplishing behaviours. Each behaviour "competes" with the others to exercise control over agent activities. Lower layers represent more primitive kinds of competences (such as obstacle avoidance in a *scout agent*) and have precedence over layers further up in the hierarchy which offer some increasingly more abstract behaviour like identifying objects, taking decisions and so forth (see figure 1.3).

A layer *L1* is unaware of the existence of a possible upper layer *L2*. *L2* in turn is able to examine data coming from *L1* and injects data into it suppressing the normal data flow. In such a schema additional "blocks" can be easily added without changes into the initial working system. All of this fosters an aspect oriented programming style [EFB01] in defining the whole agent behaviour in which every layer has an asymmetric knowledge about the other one.

A subsumption architecture offers an horizontal slicing of control aspects which contrasts classical functional decompositions [Bro86] in which every module (or layer) has to be entirely "traversed" in order to take an action, for instance, after sensing some changes into the environment.

By using this approach the resulting systems are very simple and efficient in terms of the amount of computational resources needed. A subsumption architecture fosters also robustness because a fail into a higher layer could not impair lower layer functionalities. However, against to these benefits, agents with many behaviours may become undoable and if more layers are used then more difficult is understanding what is going on.

Another different way for building reactive agents bases on the *situated automata approach* [RK95]. By using this approach an agent is viewed as an automaton which, on sensory input, takes an action and changes its current state to a new state. Each agent is first specified in terms of a declarative formalism and then this specification is translated into a low-level digital ma-

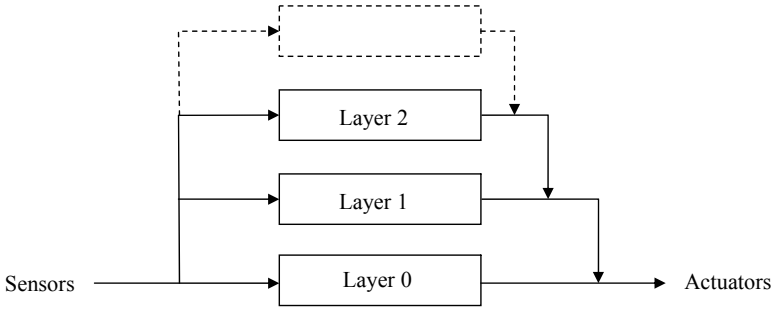


Fig. 1.3. Subsumption architecture layers

chine which satisfies the declarative specification. The machine can operate in a provably time-bounded fashion because it does not use any symbol manipulation. At runtime the agents are purely reactive.

The situated automata approach is not intended to replace deliberative architectures. It allows one to describe the informational content of an agent's computational state in a semantically rigorous way without requiring a commitment to conventional run-time symbolic processing. This is suitable to the development of so called *situated agents* whose characteristic is in their close interaction with the environment in which they are situated. Adopting this view, the fundamental phenomena to be explained is not the ″reasoning″ but the mutual constraints exhibited among parts of a physical system over time. Further reading may be found in [RK95, WJ95, BMS02].

1.3.3 Hybrid Architectures

They rely on the assumption that neither a completely deliberative nor completely reactive approach is adequate for building agents. An *hybrid approach* results from combining best elements of both reactive and declarative systems.

From this perspective an agent can be composed of two (or more) subsystems: a deliberative one, containing a *symbolic world model*, which develops plans and takes decisions, and a *reactive* one which is capable of reacting to events that occur in the environment without engaging complex reasoning. The reactive component may exhibit a kind of precedence over the deliberative one, so that it can provide a rapid response time to important environmental events. This kind of structuring leads naturally to the idea of a layered architecture where major issues are related to better understanding relationships and interdependences occurring among layers.

One of the most known hybrid architecture (see [WJ95]) is the *Procedural Reasoning System* (PRS) which is considered the best established agent architecture currently available. Originally described in [GL87], this architecture has progressed from an experimental LISP version to a fully fledged C++ implementation known as the *distributed Multi-Agent Reasoning System* (dMARS), which has been used in most significant multi-agent applications [GR96]. As an example, *Oasis* is a system for air traffic management that addresses issues of aircraft scheduling, comparing actual progress with established sequences of aircrafts, estimating delays, and notifying controllers to correct deviations. Oasis successfully completed operational tests at Sydney Airport in 1995.

The PRS architecture has its conceptual roots in the belief-desire-intention (BDI) model. In tandem with the evolution of the PRS architecture into an industrial-strength production architecture, the theoretical foundations of the BDI model have also been closely investigated (see [RG95]).

Using dMARS architecture (see figure 1.4), the BDI model is made *operational* through *plans*. Each agent has a set of plans (*plans library*) specifying

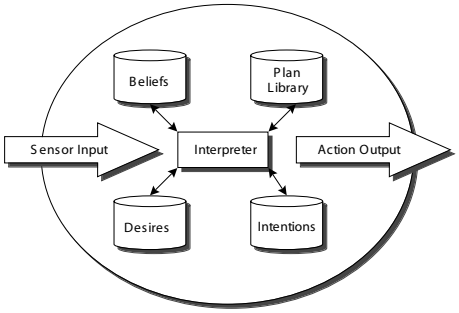


Fig. 1.4. dMARS architecture

courses of actions that may be undertaken by an agent in order to achieve its intentions. An agent plan library represents its *procedural knowledge*.

Each plan contains several components: (i) a trigger or *invocation condition* which specifies the circumstances under which the plan should be considered, (ii) a plan context, or *pre-condition*, specifying the circumstances under which the execution of the plan may commence, (iii) a *maintenance condition* characterising the circumstances that must remain true while the plan is executing, (iv) a *body* defining a potentially quite complex course of actions, which may consist of both goals (or subgoals) and primitive actions. Primitive actions can be thought of as procedure calls. An *interpreter* (figure 1.4) is responsible for managing the overall operations of the agent.

Further details about dMARS architecture may be found in [DLG+04].

1.4 Foundation technologies

As stated in previous sections, agent based computing appears to be very useful in many different and heterogenous contexts. For instance, the usage of agent systems to *simulate* real-world domains may provide answers to complex physical or social problems otherwise very hard do obtain or practically unobtainable (e.g., modelling the impacts of climate changes on various biological populations, or modelling the impact of public policy options on social or economic behaviours). However the characteristics making the agent metaphor very appealing require to be supported and implemented. As a consequence many links exist between the agent metaphor and other disciplines considered as sources of so called *"foundation"* technologies exploitable for supporting mobility, social ability and so forth. A survey on some of these topics is reported in [LMP04, LMSW05].

1.4.1 Agent platforms

An *agent platform* can be viewed as *middleware layer* providing a platform-independent execution locus in which software agents may act. It provides

support for software agents to execute, to manage their execution, to access system resources, and to guarantee integrity and protection of agents and the platform itself. Agent platforms also support migration, naming, location and communication services. They may enable methodologies and offer tools which aid in developing a multi-agent system.

This section may be considered as a complement to section 1.3 where agent architectures are introduced and where an *architecture taxonomy* is reported on the basis of the agent rational behaviours. Here agent platforms are differentiated on the basis of offered functionalities.

A very large number of agent platforms, coming either from academia or industrial contexts and often developed using Java technologies, exist today. Attempts to standardise agent platforms have resulted in the establishment of two main standards, MASIF [MBB+99] and FIPA [FIP]. Agents belonging to specific platforms affiliating to these standards can collaborate in achieving common goals via inter-agent messaging. However, agents can only migrate to and execute in remote sites only if the site's hosts run a compatible agent platform and the agent has the credentials to surpass the site security checks. This means that the concept of agents freely roaming Internet sites performing tasks on behalf of the user is yet unrealistic.

Jade [BR01] from TLAB, is an open source platform for peer-to-peer agent based applications fully implemented in Java. Jade simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications [FIP] and through a set of graphical tools that supports the debugging and deployment phases. System configuration is achieved by means of a remote GUI.

JACK [HRHL01] is an example of commercial agent platform. It incorporates a suite of graphical tools, targeted at analysts as well as programmers, suitable for building, running and integrating commercial-grade multi-agent systems using a component-based approach. Agent development rely on the JACK Agent Language which extends Java with agent-oriented concepts.

The JAMES platform [SBS00] instead, pays attention to agent system fault tolerance issues [OWB04, FD02]. It provides schemes for error detection, checkpointing and restarting failed agents, as well as a reliable migration protocol that deals with network partitioning.

Notable cornerstones of agent platform work include Actor Foundry [VA01a] based on the Gul Agha Actor model [Agh86], Aglets from IBM [DM98], Mole from the University of Stuttgart [BHR+02], Concordia [WPW+97], TACOMA from the Universities of Tromso and Cornell [JLvR+02], Grasshopper from IKV [BBCM98], as well as D´Agents from Dartmouth College [GCK+02b] and Voyager [Gla98], though there are many many others.

1.5 Mobile agents

Mobile agents [Pic01, Ple99] provide a valuable alternative to the traditional client/server programming model (among others), because they provide a uniform paradigm for distributed systems. Nevertheless, mobile agent technology is not yet widespread in today's applications. Conventional distributed systems typically assume a static configuration of the environment where the distributed application executes. Communication among a set of hosts is enabled by physical links whose configuration is fixed and statically determined. Similarly, the various portions of the distributed applications that run on the nodes of the system are typically bound to such nodes for their whole life. This view is being challenged by technical developments that introduce a degree of *mobility* in the distributed system. The so called *code mobility* allows for the code and possibly the state of an executing program to be *migrated*, in part or as a whole, at run-time. The paradigm of mobile code generalizes this concept by performing changes along two orthogonal axes:

- Where is the know-how of the service located?
- Who provides the computational resources?

Depending on the choices made on the server and client sides, the following additional paradigms can be identified [FPV98]]:

- **Remote Evaluation (REV)**. In the REV paradigm a component A sends instructions specifying how to perform a service to a component B. The instructions can, for instance, be expressed in Java bytecode. B then executes the request using its resources.
- **Code on Demand (CoD)**. In the CoD paradigm the same interactions take place as in REV. The difference is that component A has the resources collocated with itself but lacks the knowledge of how to access and process these resources. It gets this information from component B. As soon as A has the necessary know-how, it can start executing.

The mobile agent paradigm is an extension of the REV paradigm. Whereas the latter focuses primarily on the transfer of code, the mobile agent paradigm involves the mobility of an entire computational entity, along with its code, state, and potentially resources required to solve the task.

1.5.1 Benefits of mobile agents

In the following some advantages of *mobile agent computing* are described:

- *Reduced bandwidth consumption.* The ability to migrate can be used to achieve co-location among agents and resources they must access in order to reduce the need for remote communication. In addition, a *semantic compression* concept can be realized. Semantic compression reduces the amount of information being transmitted by filtering it at the source,

based on its content. This is dramatically different from the client/server approach where an unfiltered over-set of information transit on the net. In mobile agent approach, a *filtering* agent is sent on the server in order to send back only the data that is really useful to client. The locality of the communicating entities allows to decrease the *latency* and save *bandwidth*. Clearly the gain in latency and bandwidth must overcome the cost of sending the agent to the server node.

- *Improved fault tolerance.* In conventional systems, a high-level interaction between a client and a server involves a series of pairwise low-level interactions under the form of requests and replies. During these interactions, the state of the overall computation is distributed. This fact heavily complicates the task of recovering from a fault, due to the distributed consensus problem. In mobile agent approach, agents embedding the code describing the whole high-level interaction can migrate on the server. Thus the state of interaction remains local, and faults can be dealt with easily.
- *Support for disconnected operations.* Mobile agents can carry out their tasks autonomously and independently of the application that dispatched them. This capability, that is at the core of many of the advantages mobile agents provide, is particularly useful in scenarios characterized by physical mobility, where the constraints posed by terminals and communication links often force the user to disconnect from the network to perform some task on behalf of a user, who meanwhile is totally disconnected. Results can be eventually gathered by the user upon reconnection.
- *Protocol encapsulation.* Mobile agents allow a packet of information to travel within the system together with the application logic needed to interpret and manipulate it. This improves the flexibility of the system, simplifying the deployment of different, co-existing policies for using data.
- *Load balancing.* An overloaded host (in terms of cpu and memory utilization) can migrate a part of his agents toward low loaded hosts.

1.5.2 Mobile agents technology

Technology has traditionally been the main focus of research on mobile agents. As a matter of fact, the term 'mobile agent' was made popular by the telescript language [Whi99], developed by General Magic in 1994. The emphasis on technology is witnessed, among the other things, by the large number of systems contained in the Mobile Agent List [Hoha], maintained at the university of Stuttgart, Germany, that provides an approximate census of existing mobile agent systems.

Mobile agent systems typically identify the agent with a unit of execution belonging to the lower layers of the virtual machine, e.g. a thread or a process. A unit of execution [FPV98] is constituted by the code governing its behaviour, the data associated with it and necessary to its computation, and by its execution state, e.g. program counter, and call stack.

Mobile agent systems allow migration of the whole unit or a part thereof, i.e. one or more of the three constituents mentioned above. The most relevant differences among existing systems lie exactly in what they allow to move, and how it is actually moved. A first distinction can be drawn based on whether the execution state is migrated along with the executing unit or not. Systems providing the former option are said to support *strong migration*, as opposed to systems that discard the execution state across migration, and are hence said to provide only *weak migration*. In this latter kind of systems, if the application requires the ability to retain the thread of control, extra programming is required in order to save manually the execution state. Instead, in systems supporting strong mobility, migration is completely transparent to the migrated entity, which resumes execution right after the migration instruction. Despite these advantages, most of the mobile agent systems support only weak mobility. Example of systems that support strong mobility are: Telescript, Tacoma [PS97], Ara [JvRS95], and D'Agents [GCK+02a].

Another dimension to understand the mechanisms supporting mobility is constituted by the strategies employed to relocate the code constituting the execution unit. Although a number of strategies are potentially meaningful and useful, the use of Java as an implementation language has often encouraged the designers of mobile agent systems towards mechanisms that are directly inspired by the Java class loader. under this scheme, only agent's root class is migrated along with the agent; after migration, additional classes needed for execution of the agent are downloaded dynamically from the agent source host, or from some other code repository on the network. This mechanism, adopted by many of the Java-based systems, notably Mole [BHR+02] and Aglets [LO98], relies on the assumption that the code repository is always available, thus implicitly neglecting one of the main advantages of mobile agents, i.e. the ability to support disconnected operations. On the other hand, other systems, e.g. D'Agent, always ship the whole code base together with the agent, thus in many cases sending also classes that are used infrequently.

Finally, the third dimension is constituted by the data the mobile agent may carry along during migration. The unit of execution running at the source is likely to contain bindings to resources (e.g. objects, files, other units) that are shared with other units on that host. To allow mobility of the executing unit requires both a mechanism and a policy to determine how these bindings are handled upon migration. In [FPV98] a number of strategies are showed. Essentially, the binding to a resource can be severed, retained, or re-estabilished with a different but compatible resource. When the binding is retained, two alternatives are possible: either the resource is migrated along with the agent, or the binding is stretched across the network by creating a *network reference* from the new host of the agent. When the binding is instead re-established to a new, compatible resource, such resource is typically constituted either by a copy of the original one, or by a stationary resource having the same type (e.g. a printer).

1.6 Distributed agent-based simulation

Simulation is the imitation of a system behaviour and structure in an experimental model in order to studying emerging system properties which are transferable to reality. In *multiagent-based simulation* (MABS) real world systems are modelled using multiple agents. A software agent is typically defined as a program that acts autonomously, communicates with other agents, is goal-oriented (pro-active) and uses explicit knowledge. The modelled system emerges by interaction of the individual agents as well as their collective behaviour. Agents typically send messages according to some communication protocol.

However, the computational requirements of simulations of agent-based systems far exceed the capabilities of conventional sequential von Neumann computer systems. Each agent is typically a complex system in its own right (e.g. with sensing, planning, inference etc. capabilities), requiring considerable computational resources, and many agents may be required to investigate the behaviour of the system as a whole or even the behaviour of a single agent. One solution to this problem is to attempt to exploit the high degree of parallelism inherent in agent-based systems to parallelise the simulation. Decentralised, event-driven distributed simulation is particularly suitable for systems with inherent asynchronous parallelism, such as agent-based systems.

1.6.1 Discrete-event simulation

In *discrete-event simulation* the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. The simulation must keep track of current simulation time, in whatever measurement units are suitable for the system being modeled. The simulation maintains at least one list of simulation events. This is sometimes called the *pending* event set because it lists events that are pending as a result of previously simulated event but have yet to be simulated themselves. An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. Discrete event techniques have been used in a number of MAS simulators. For example, the SPADES system [BGFL94] and the JAMES system [SU00] and RePast [Hohb].

1.6.2 Distributed simulation

Distributed simulation addresses two key problems of existing MAS simulations and simulators [Log07, PS09]: scalability and simulation re-use. Distributed simulation (along with conservative or optimistic algorithms for time coordination and synchronization in the-large[Fuj00]) exploits the natural parallelism of MAS. Simulation components can be distributed so as to make the

best use of available computing resources, allowing agent researchers and developers to run agent simulations in less time and/or investigate multi-agent systems which are simply too large to be effectively simulated on a single computer. Distribution also promotes inter-operability of simulators and simulation components. No one simulator or testbed is, or can be, appropriate to all agents and environments. Investigating a particular problem therefore frequently entails the development of a new simulation. The effort required to develop a new simulation from scratch is considerable. There is therefore a strong incentive to reuse existing simulation components, toolkits and testbeds for a new problem.

In the last years there was an explosion of interest in distributed simulation as a strategic technology for linking simulation components of various types at multiple locations to create a common virtual environment.

1.6.3 The problem of shared state

While conventional distributed simulation can bring real benefits from an inter-operability point of view, the speedups that can be attained in practice (particularly for situated MAS) are often rather limited. The simulation of situated agents (e.g., robots situated in a physical environment, or characters in a computer game or interactive entertainment situated in a virtual environment) presents particular challenges which are not addressed by standard parallel discrete event simulation (PDES) models and techniques. While the modelling and simulation of agents, at least at a coarse grain, is relatively straightforward, it is harder to apply conventional PDES approaches to the simulation of the agents environment. In a conventional decentralised event-driven distributed simulation, the model is divided into a network of Logical Processes (LPs). Each LP maintains its own portion of the simulation state and LPs interact with each other in a small number of well defined ways. The topology of the simulation is determined by the topology of the simulated system and its decomposition into LPs, and is largely static.

In contrast, the interaction of agents in a situated MAS is often hard to predict in advance. Different kinds of agents have differing degrees of access to different parts of the environment at different times. The degree of access is dependent on the range of the agent sensors (read access) and the actions it can perform (write access). For example, what a mobile agent can sense is a function of the actions it performed in the past which is in turn a function of what it sensed in the past. As a result, it is difficult to predict which parts of the simulation state an agent can or will access without running the simulation. This makes it hard to determine an appropriate topology for a MAS simulation a priori, and simulations of MAS typically have a large shared state, the agents environment, which is only loosely associated with any particular process. This shared state can form a bottleneck, limiting the speedups that can be attained.

1.7 Conclusions

This chapter surveys basic concepts of *agent oriented computing* and get into focus about property of mobility and the use of agents in distributed simulation. Even if agent computing is considered useful for developing complex software systems, it is worth of note that adopting a technology in place of another is a very important issue [WJ98]. For instance, agent computing is not suitable in application domains where an high degree of orchestration, i.e., a strong centralized coordination, is required. Moreover, agents usually involves multiple threads which intrinsically introduce uncertainty in establishing temporal behaviour of a software system.

Nowadays the adoption of agent technologies has not yet entered in the mainstream of commercial and industrial organisations [LMSW05]. Probably this is due to research area of agent technology which is still only in its infancy and to the lack of proven methodologies, tools and widely accepted standards.

Much of the standardisation effort in the agent community fall in the *Foundation of Intelligent and Physical Agent* (FIPA) and the *Object Management Group* (OMG), which are the premier agent standardisation bodies.

In particular, FIPA is a promising organisation that promotes agent-based technology and the interoperability of its standards with other technologies. FIPA was officially accepted by the IEEE as its eleventh standards committee on June 2005.

Current research efforts (see [LMP04, LMSW05]) are mainly devoted to cover previously stated holes. In particular, standardisation efforts appear to be very critic. Extending links with other disciplines (biology, economics, sociology, game theory, and so forth), whose techniques could be applied to agent systems, represents another main issue.

2

M&S formalisms, middleware, tools

2.1 Introduction

This chapter is organized into three parts dealing with some M&S formalisms, some standard or emerging middleware, some agent-based toolkits and related concept.

The size and complexity of systems which are usually modelled as discrete event systems (DESs) (e.g. communication networks, biological systems, weather forecasting, etc.) is ever increasing. Modelling and simulation of such systems is challenging in that it requires suitable specification languages and efficient simulation tools.

Many languages exist to cope with complexity issues. In the first part two formalisms are discussed. First DEVS [ZPK00] are briefly reviewed which are capable of ensuring the necessary rigor in the modelling phase and in particular Parallel DEVS (P-DEVS) which is highly recognized as a reference point in M&S. The second formalism concerns *statecharts* [Har87, BRJ99] that is an evolution of finite state machines that offer a nested state structure and other powerful mechanisms for specifying complex system.

In the second part of the chapter some common and standard middleware are examined which can effectively be used for modelling and distributed/-parallel simulation. More specifically the IEEE standard HLA/RTI and the emerging Terracotta architecture are discussed.

In the last part of the chapter some fundamental tools supporting agent-based software engineering are presented. In particular the focus is on the RePast toolkit which is representative of current state-of-art of agent-based M&S tools. In addition basic concepts of aspect oriented programming (AOP) are discussed, which are a key for solving some software engineering problems.

2.2 Formalisms

2.2.1 DEVS

The Discrete-Event System Specification (DEVS) is a well known formalism [ZPK00] which enables the specification of discrete-event systems through mathematical concepts borrowed from the systems theory. A system specification is characterised by a time base and proper sets of inputs, outputs and states, and functions for determining the next state and outputs on the basis of current state and input. The strength of the formalism is in the rigorous and systematic way through which the modeller has to proceed during the abstraction of the parameters and behaviour of the system model. After the abstraction phase, the specification can then be mapped onto a implementation framework for simulation and output analysis. In the classic DEVS formalism, *Atomic DEVS* captures the system behaviour, while *Coupled DEVS* describes the structure of system.

In the following the extension to classic DEVS named Parallel DEVS is considered which is more expressive in capturing simultaneity and parallelism in component behaviour.

Atomic DEVS

In P-DEVS an atomic component specification (Atomic DEVS) is a structure M defined as

$M =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$ where

- X is the set of input values
- S is a set of states
- Y is the set of output values
- $\delta_{int} : S \to S$ is the *internal transition* function
- $\delta_{ext} : Q \times X^b \to S$ is the *external transition* function, where
  $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is is the set of *total states*
  e is the is the *elapsed time* since last transition
  X^b denotes the collection of *bags* over X (in a bag some elements may occur more than once)
- $\delta_{con} : Q \times X^b \to S$ is the *confluent transition* function
- $ta : S \to \Re^+_{0,\infty}$ is the time advance function

The interpretation of the elements of M can be summarized as follows. At any time the system is in some state $s \in S$. The system can remain in s for the time duration (dwell-time) $ta(s)$. $ta(s)$ can be 0, in which case s is said a *transitory state*, or it can be ∞, in which case it is said a *passive state* because the system can remain forever in s if no external event interrupts. Provided no external event arrives, at the end of (supposed finite) time value $ta(s)$, the system moves to its next state $s' = \delta_{int}(s)$ determined by the internal transition function δ_{int}. In addition, just *before* making the internal transition, the system produces the output computed by the output function $\lambda(s)$. During

its stay in s, the system can receive an external event x which can cause s to be exited earlier than $ta(s)$. Let $e \leq ta(s)$ be the elapsed time since the enter time in s (or, equivalently, the time of last transition). The system then exits state s moving to next state $s' = \delta_{ext}(s, e, x)$ determined by the external transition function δ_{ext}. As a particular case, the external event x can arrive when $e = ta(s)$. In this case two events occur simultaneously: the internal transition event and the external transition event. The next state s', in this *collision* situation, is determined by the confluent transition function δ_{con}. The default behaviour of δ_{con} first applies the internal transition function and then the external transition function. This behaviour, though, can be redefined. After entering state s', the new time advance value $ta(s')$ is computed and the same story continues. The presence of confluent transition function together with the presence of bags instead of normal sets qualify model as *Parallel DEVS*. P-DEVS allows multiple components to be activated and send their output to other (influences) components at the same time. It is worth noting that there is no way to generate an output directly from an external transition. An output can only occur just before an internal transition. To have an external transition cause an output without a delay, a transitory state can be entered from which the exiting internal transition is preceded by the generation of the output value.

In reality inputs arrive to a component through input ports and outputs are trasmitted externally through output ports. An input is thus a pair $< inp, x >$ of an input port *inp* and an event x. Similarly, an output is a pair $< outp, y >$ of an output port *outp* and symbol value y. Ports are topological entities which decouple components and favour reconfiguration.

Coupled DEVS

Basic models may be coupled to form a *coupled model*. A coupled model tells how to several component models connect each other and with the external environment to form a new model. A coupled model can itself be employed as a component in a larger model, thus giving rise to hierarchical construction. A coupled model contains the following information:

- a set of components
- a set of input ports from which external events are received
- a set of output ports through which external events are sent

These components can be synthesized together to create hierarchical models having external input and output ports. The coupling specification (see Fig. 2.1) consists of:

- the *external input coupling* which connects input ports of the coupled model to one or more of the input ports of internal components, this directs inputs received by the the coupled model to designated component models

- the *external output coupling* which connects output ports of internal components to output ports of the coupled model. Thus when an output is generated by a component it may be sent to a designated output port of the model and thus transmitted externally
- the *internal coupling* which connects output ports of internal components to input ports of other internal components. When an input is generated by a component it may be sent to the input ports of designed components (in addition to being sent possibly to an output port of the component model

A coupled model is equivalent to a basic model in the DEVS formalism and can itself be employed in a larger coupled model. This shows the formalism is closed under coupling as required for hierarchical model construction. Expressing a coupled model as an equivalent basic model hides logic by which components interact to yield an overall behaviour.

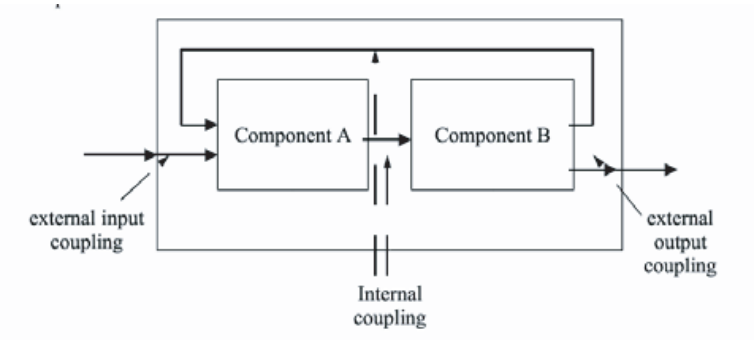


Fig. 2.1. Schema of a coupled model

DEVS tools

In the last years many toolkits and libraries have been developed for the DEVS formalism. One of the most known is DEVSJAVA [ZS03]. The software is written in Java and supports parallel execution on a uni-processor. It supports higher-level, application specific modeling. Models in DEVSJAVA can also be easily mapped to DEVS/HLA or DEVS/CORBA for distributed execution in combined logical/real-time settings. Other Java DEVS tools are: JAMES II [JAM] that provides support for many different formalisms, among these variants of DEVS formalism (e.g., PDEVS) and JDEVS [FDB02]. DEVS/C++ [ZMKK96] based on the parallel DEVS formalism, is a modular hierarchical discrete event simulation environment implemented in C++. Other C++ DEVS tools are: PowerDEVS [KLP03] , ADEVS [ADE] and CD++ [CDP]. Besides tools is worth to note some important projects related to DEVS. DEVS/HLA [ZBC+98] is a project aimed at demonstrate how an HLA-compliant

DEVS environment can significantly improve the performance of large-scale distributed modeling and simulation exercises. This project has its foundation on the High Level Architecture. Another project, SimBeans [PSS99], employs the system theoretic simulation modeling methodology of DEVS as formal, mathematical foundations for modular, hierarchical modeling and simulation and a component based software architecture based on Java and JavaBeans.

An integrated approach (eUDEVS) for M&S was proposed in [RMDLCMZ09]. It characterises for the use of the DEVS formalism for making an UML specification executable and analyzable by simulation. eUDEVS permits the modeller to start a project with UML diagram models which allow to abstract system structure and behaviour. UML diagrams, including statecharts, are then mapped into the specific class of DEVS models named Finite and Deterministic DEVS (FD-DEVS) which are represented in an XML format. A transformed UML specification can be simulated using e.g. DEVS-JAVA

2.2.2 Statecharts

Statecharts [Har87, BRJ99] are an extension of classical state transition diagrams. Statecharts are used to model the dynamic behaviour of a component or a system. Statecharts are specifically used to define state-dependent behaviour, or behaviour that varies depending on the state in which a model element is in. A statechart consists of states, linked by transitions. A state is a condition of a system in which it performs some activity or waits for an event. A transition is a relationship between two states which is triggered by some event, which performs certain actions or evaluations. The basic difference with classic state transition diagrams consists in the possibility of nesting a sub automaton within a (macro) state thus encouraging step-wise refinement of complex behaviour. In addition, a macro state can be *and-decomposed* for supporting a notion of concurrency. Statecharts have been successfully applied to the design of reactive event-driven real-time systems [HP98, SR98, FNP06], as well as to modelling and performance analysis, e.g. [VCA02, VCAA06]. An example of a statechart diagram is shown in Figure 2.2.

States

A *state* is a condition of a modelled system in which it performs some activity or waits for an event. A state has several properties:

- *name*: A textual string which distinguishes the state from other states; a state may also be anonymous, meaning that it has no name.
- *entry/exit actions* executed on entering and exiting the state.
- *internal transitions* i.e. transitions that are handled without causing a change in state.
- *substates*: the nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates.

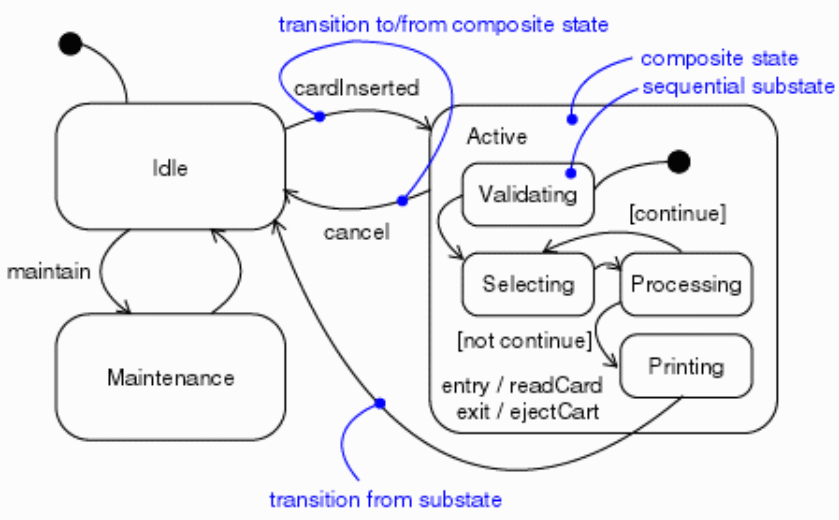


Fig. 2.2. Statechart example

States are denoted by a rounded square symbol. As depicted in Figure 2.2, there are two special states that may be defined for a statechart. The *initial default state* indicates the starting place for the statechart or substate. An initial state is depicted as reached by a pseudo-transition leaving from a filled black circle. *final state* indicates the completion of the execution of the state machine or that the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle. Initial and final states are really pseudostates. Neither may have the usual parts of a normal state, except for a name. A state of statechart can recursively be decomposed into a set of sub states, in which case it is said to be a *macro* state. A state that is not decomposed is said to be a *leaf* state. The root state of the decomposition tree is the only one having no parent and it is referred to as the *top* state. This nested structure of states implies that system modelled by a statechart, at a given point in time, finds itself simultaneously in a set of states that constitutes a path leading from one of the leaf states to the top state. Such a set of states is called a *configuration* [HN96]. A configuration is uniquely characterised by the only leaf state which it contains.

Statecharts admit two types of state decomposition: *or*-decomposition and *and* decomposition [HP98]. In the former case a state is split into a set of sub states which are in an "exclusive-or" relation, i.e. if at a given time the modelled system is in a macro state it is also in exactly one of its sub states. In the other case sub states are related by logical "and", i.e. if the modelled system is in a macro state it is also in *all* of its direct sub states, each of which acts as an independent concurrent component.

Transitions

A *transition* is a relationship between two states indicating that system in the first state will perform certain actions and enters a second state when a specified event occurs and stated conditions are satisfied. On such a change of state, the transition is said to 'fire'. Until the transition fires, the object is said to be in the 'source' state; after it fires, it is said to be in the 'target' state. A transition has several properties:

- *source* state: the state affected by the transition; in a source state, an outgoing transition may fire on receiving the trigger event of the transition provided the guard condition, if any, is satisfied.
- *event trigger*: the event that makes the transition eligible to fire (providing its guard condition is satisfied) when received by the object in the source state.
- *guard condition*: a boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates to *true*, the transition is firable; if the expression evaluates to *false*, the transition is disabled. If there is no other transition that could be triggered by the same event, the event is lost.
- *action*: an executable atomic computation associated with the transition.
- *target* state: the state that is active after the completion of the transition.

A transition may have multiple sources, in which case it represents a join from multiple concurrent states, as well as multiple targets, in which case it represents a fork to multiple concurrent states. An event is an occurrence of a stimulus that can trigger a state transition. Events may include signal events, call events, the passing of time, or a change in state. A signal or call may have parameters whose values are available to the transition, including expressions for the guard conditions and action. It is also possible to have a triggerless transition, represented by a transition with no event trigger. These transitions, also called *completion* transitions, is triggered implicitly when its source state has completed its activity. A guard condition is evaluated after the trigger event for the transition occurs. It is possible to have multiple transitions from the same source state and with the same event trigger, as long as the guard conditions don't overlap. A guard condition is evaluated just once for the transition at the time the event occurs. The boolean expression may reference the state of the object. An action cannot be interrupted by an event and therefore runs to completion. This is in contrast to an activity, which may be interrupted by arrival of other events. Actions may include operation calls, the creation or destruction of objects, or the sending of a signal to another object. In the case of sending a signal, the signal name is prefixed with the keyword 'send'. An internal transitions allow events to be handled within the state without leaving the state, thereby avoiding triggering entry or exit actions. Internal transitions may have events with parameters and guard conditions. State transitions are represented by edges with arrows.

Each transition is labelled by *ev[guard]/action* where *ev* is the trigger, *guard* the guard condition, and *action* the action to perform.

History pseudo-states

When a transition enters a composite state, the action of the nested state machine starts over again at the initial state (unless the transition targets a substate directly). *History* states allow the state machine to re-enter the last substate that was active prior to leaving the composite state. An example of history state usage is presented in Figure 2.3.

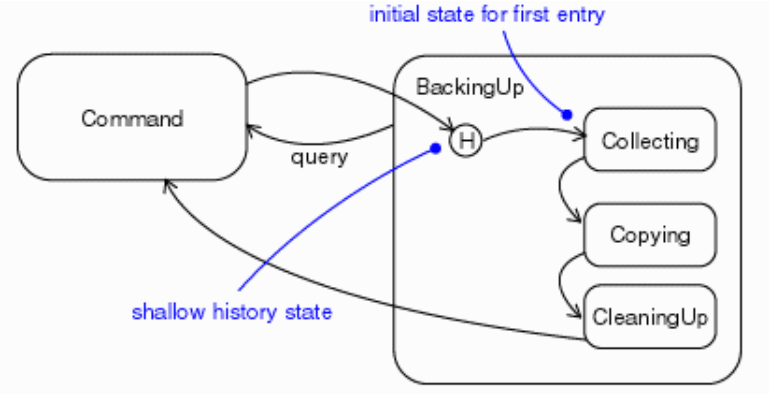


Fig. 2.3. History connector

This pseudo-state is graphically identified by an *history connector* or *H*-connector. Such a connector is depicted as a small circle containing an H (shallow history) or an H* (deep history), and it is always inside the boundary of a compound state. The difference between shallow and deep history concerns composite states with nesting level greater than one. With shallow history, when a macro state is left, only the directed substate is "remembered". With deep history, instead, all configuration until the leaf state is kept in memory. More specifically, Let S be the destination state of a transition tr. The configuration which is assumed as consequence of firing tr depends on the way tr reaches S:

- If S is a leaf state the new configuration is the only one that contains S.
- If S is a macro state and tr ends on its border, the next configuration corresponds to the destination state being the initial state of S.
- If S is a macro state and tr ends on a shallow history connector, the next configuration corresponds to the destination state being the state that is the history of S
- Finally, if S is a macro state and tr ends on a deep history connector, the configuration depends on the nature of the state D which is currently

history of S. If D is a leaf state, the configuration will be the only one that contains D, otherwise the configuration corresponds to the case tr would end on a deep history connector of D.

2.3 Middleware

2.3.1 HLA-RTI

HLA (High Level Architecture) [KWD99, HLA], originally developed under the leadership of the Defense Modeling and Simulation Office (DMSO), facilitates reuse and interoperability among (possibly heterogeneous) simulations [HLA]. HLA was accepted as an IEEE standard (IEEE 1516) in September 2000. HLA supports component-based simulation development, where the components are referred as *federates* that interact each other in order to form a combined simulation system known as *federation*. More specifically, a federation contains: the federates, a common object model specifying data exchanged between federates in federation, called *Federation Object Model* (FOM), and a supporting software for federation execution called *RunTime Infrastructure* (RTI).

HLA defines a software architecture, not an implementation. Shaw and Garlan [SG96] defines a software architecture as:

...Abstractly, software architecture involves the description of elements from which systems are built, interaction among those elements, patterns that guide their composition, and constraints on those patterns.

HLA as architecture defines as its *elements* the federates and an execution supporting software RTI. HLA *interactions* concern rules and interface specification defining interactions between federates and RTI, and between federates (always mediated by RTI). The *Object Model Template* is a metamodel for all FOM (*Federation Object Model*), that is, it defines the structure of every valid FOM. Finally, the allowed *patterns* of composition in the HLA are constrained by the rules and defined in the interface specification.

The RTI provides the infrastructure that allows federates belonging to the same federation to communicate with each other in a distributed environment. Figure 2.4 shows how the federates interact with the RTI. Federates do not directly communicate with each other but always through the RTI.

Federates communicate with the RTI through a well defined interface. This interface, the *RTI Ambassador* interface, defines several services which a federate can call upon the RTI. A federate is only allowed to communicate through this service interface with the RTI. Callbacks from the RTI to the federate are defined through the *Federate Ambassador* interface. The RTI can call services on this interface to interact with the federates. Figure 2.5 shows the communication channels between a federate and the RTI.

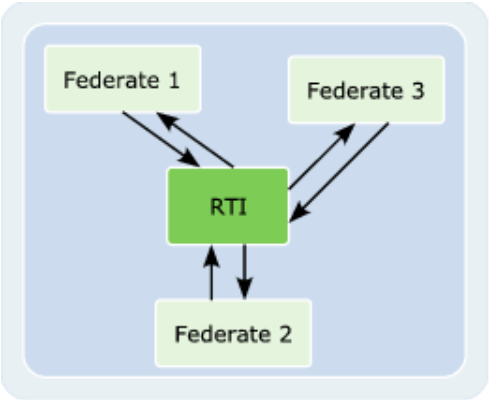


Fig. 2.4. Basic structure of a federation

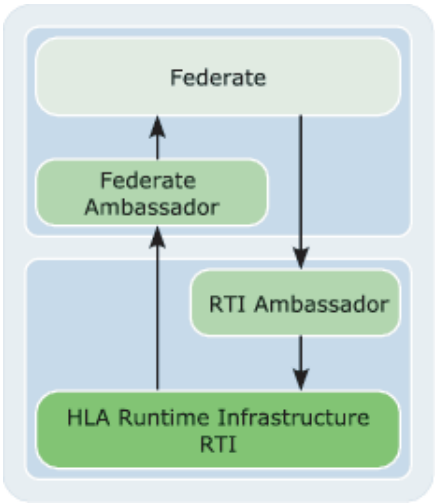


Fig. 2.5. Interfaces between RTI and federates

FOM

Executing a federation requires designing a *Federation Object Model* (FOM) and using the services of the RTI middleware. The FOM, specified in a Federation *Execution Data file* (FED file), defines types and relationships of the data exchanged between federates. In particular, the FOM introduces a set of **object classes** and a set of **interaction classes**. Object classes, along with their attributes, are created by federates and constitute a *persistent state* of a simulation. An interaction is made up of parameters and models an *event occurrence*. An interaction is consumed after being received. Object and in-

teraction classes are organized into separate hierarchies. A class inherits the attributes or parameters of its superclasses.

RTI

The *runtime infrastructure* (RTI) is implemented as a distributed component in itself, split into a *Central RTI Component* (the CRC) and one or more *Local RTI Components (LRC)*. Figure 2.6 shows these components of the RTI.

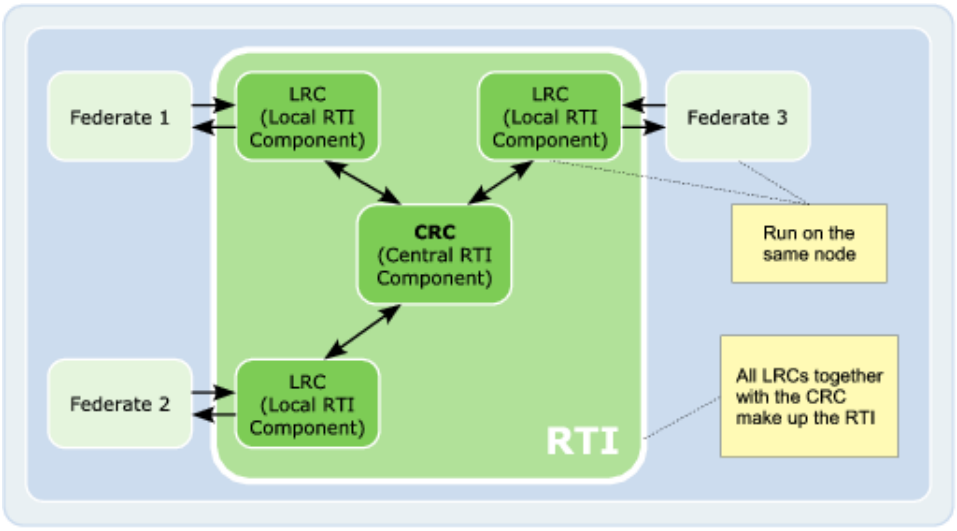


Fig. 2.6. RTI components

Each of the RTI components (CRC and LRC) can run on a different node and each RTI component can communicate with any other component. This allows the LRC to directly communicate with another LRC but it also allows the CRC to communicate with any LRC. Note, however, that a federate never communicates with the CRC directly but always through the LRC. In practice this means that the LRC exposes the RTI Ambassador interface to the federate. Accordingly, the CRC does not communicate directly with a federate either but again through the LRC only. The LRC translates calls from the CRC to calls on the Federate Ambassador interface on the federate.

RTI services

Communication between federates and federation is mediated by *RTI services*. Communication between a federate and RTI is structured following a request/-callback mechanism through their interfaces (RTI Ambassador and Federate Ambassador). From this point of view, a publish/subscribe design pattern is used. For example, when a federate updates the value of a published attribute,

the RTI ensures the update is automatically notified, through RTI initiated callbacks, to federates which subscribed to the attribute. RTI provides services to federations that fall into six groups that are defined by similarity of interest:

- **federation management** these services manage a federation in two ways: (i) by defining a federation execution in terms of existence and membership, (ii) by accomplishing federation-wide operations.
  To define a federation, there are services to **create**/**destroy** a federation execution and to permit a federate to *join* the execution or *resign* from it. Federation-wide operations include the coordination of federation saves (checkpoints) and restore. There are also services to allow a federation to define and meet a federation-wide **synchronization points**.
- **declaration management** The HLA is characterized by an implicit-invocation style of data exchange. Federates don't send data to other federates by name; they make it available to the federation, and the RTI ensures its delivery to interested parties. The *declaration management* services are the way federates declare their intent to produce (**publish**) or consume (**subscribe**) data. The RTI uses these declaration for routing data, transforming data, and interest management.
- **object management** these services are those used for actual exchange of data. A federate uses services from this group to **send** and **receive** interactions or to **register** new instances of an object class and to **update** its attributes. Other federates will have services from this group invoked on them to **receive** interactions, **discover** new instances, and receive updates of instance attributes.
- **ownership management** In HLA terms, simulating an entity means furnishing values for instance attributes. The *ownership management* services implement the HLA's notion of responsibility for simulating an entity. They allow that responsibility to be shared or transferred among federates. The RTI ensures that at most one federate at a time owns a given instance attribute.
- **time management** With federates executing in their own thread of control, the proper ordering of events between federates is a significant problem to be solved. In the HLA, ordering of events is expressed in "**logical time**". The RTI's *time management* services do two things: (i) allow each federate to advance its logical time in coordination with other federates; (ii) control and delivery of time-stamped events so the federate need never receive events from other federates in the "past", that is, events with logical times less than its logical time.
  The RTI allows a federate to choose the degree to which it participates in time management. A federate may be *time constrained*, in which case its advance of logical time is constrained by other federates. A federate may be *time regulating*, in which case its advance of logical time regulates other federates.

Services of this group allow: both **time-stepped simulation** and **event-driven simulation**. They also permit: **conservative**, **optimistic**, and **episodic** simulation.

- **data distribution management** Services of this group control the *producer/consumer* relationships among federates. Whereas the declaration management services manage those relationships in terms of interaction and object classes, *data distribution management* manages in terms of object instances.

2.3.2 Terracotta

Terracotta [ZBB+08, Ter] is a *transparent clustering service* for Java applications. It can also be referred to as *JVM-level* clustering. Terracotta JVM-level clustering technology aids applications by providing a simple, scalable, highly available world in which to run.

JVM-level clustering simplifies development of enterprise Java systems by enabling applications to be deployed on multiple JVMs, yet interact with each other as if they were running on the same JVM.

Terracotta is based on extending the Java Memory Model of a single JVM to include a cluster of virtual machines such that threads on one virtual machine can interact with threads on another virtual machine as if they were all on the same virtual machine with an unlimited amount of heap. This approach aids the programming model of applications clustered using Terracotta to be the almost same to that of an application written for a single JVM.

Terracotta provide a *global* (or *super*) heap in the form of a cache-like *network-attached memory* (NAM) managed by Terracotta server. NAM can host arbitrarily large shared object graphs which are kept consistent and automatically replicated/updated, at access time, in the local heap memory of clustered and interested JVMs.

There is no developer API specific to Terracotta. Indeed, NAM is controlled through an XML configuration file (tc-config.xml) specifying which objects have to be shared and how they have to be managed.

The real work on global shared objects occurs behind the scene through *aspect oriented programming* (AOP)(see 2.4.2) mechanisms and weaving, and precisely through *bytecode injection* at class loading time. In other terms, any access to shared data is intercepted and handled by injected bytecode. All of this ensures transparency and makes it simple to switch the execution environment of an application from a single JVM to a cluster of JVMs.

Terracotta is made up of libraries installed next to any JVM that is to join a cluster, and a *Terracotta server* that is a separate Java process, as shown in figure 2.7.

Benefits of Terracotta can be grouped in *key-words* that are explained below:

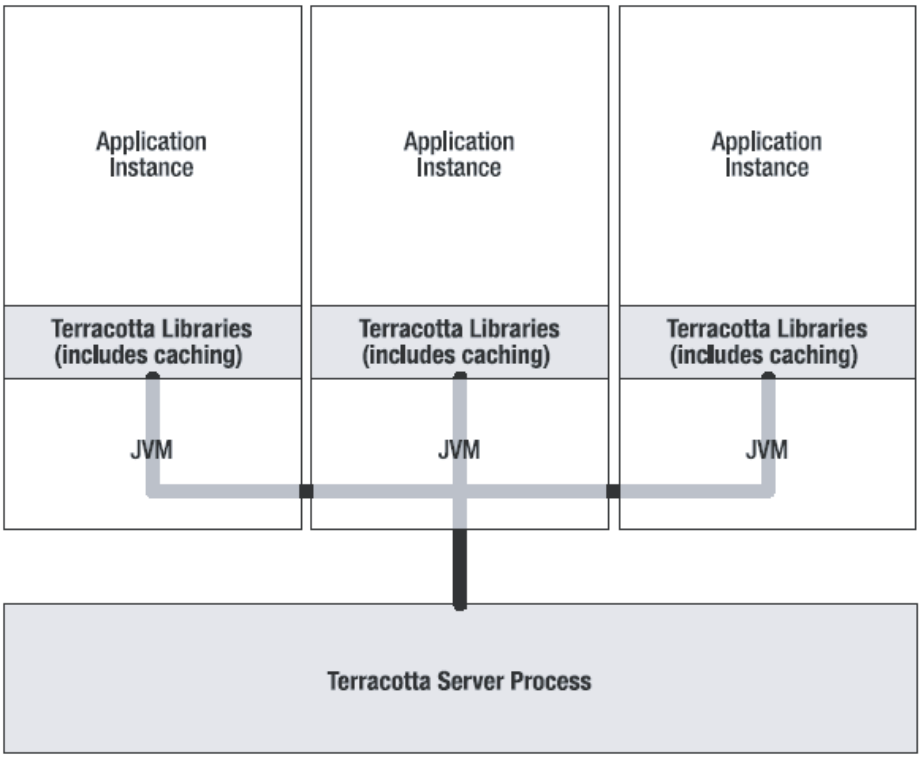


Fig. 2.7. Terracotta architecture

- **Transparency**: to users, transparency means that an application written to support Terracotta will still function as implemented with zero changes when Terracotta is not installed.
- **Clustering Service**: while clustering refers to servers talking to each other over a network, a clustering service refers to technology that allows to take an application written without any clustering logic or libraries and spread it across servers by clustering in the JVM, below the application
- **Simplicity**: Complexity and its antithesis simplicity refer to the changes that developers have to make along the road to building scalable applications.
- **Scalability**: is important to save money by starting small and growing only when demand requires. Scalability is the result of low latency and high throughput. An application can be very quick to respond but might only handle one request at a time (low latency and low throughput). Alternatively, an application can be very slow to respond but handle thousands of concurrent requests (high latency and high throughput)

- **Availability**: Availability means that every piece of shared data must get written to disk. If the data center in which the application cluster is running on top of Terracotta loses power, nothing will be lost when the power is restored.

Clustered objects

Objects that have to be clustered are specified in a configuration file (tc-config.xml). In general, objects refer each others to form object graphs, the top level of clustered object graphs are said **roots** objects. A root is any field in any class declared in the Terracotta configuration as the beginning of a clustered object graph. The first time a value is assigned to a field declared as a root, Terracotta goes through the process of turning that object into a root. Terracotta *traverses the object graph*, transforming each of those objects, including the root itself, into a clustered object. Roots are declared in the Terracotta configuration by name, and it is possible to create as many roots as needed.

Fields that are declared as roots assume special behavior. This is the one area of Terracotta that diverges significantly from regular Java semantics. The first time a JVM assigns a value to a root (which assigns it for the lifetime of the cluster), the root is created in the Terracotta cluster. Once assigned, the value of a root field may never be changed, after the first assignment all subsequent assignments to fields declared to be that root are ignored.

Roots persist beyond the scope of a single JVMs life cycle, they are sometimes called *superstatic* fields, i.e. while static field references have the same life cycle as the classes they are declared in, roots have the same life cycle as the virtual heap (this is true no matter what the scope modifier of the root field is).

While the reference of the root field itself cannot be changed, the object graph of that root object can. When an object becomes clustered, all of its field data is collected and stored in the Terracotta virtual heap. That object is also assigned to a cluster wide unique object ID that is used to uniquely identify the object in the Terracotta virtual heap. At the appropriate time, all of this object data is sent to the Terracotta server, and thereafter, it is managed by Terracotta as a clustered object.

The Terracotta client libraries running in application JVMs are free to manipulate the physical heap in the client JVMs. This means that not every object on the virtual heap needs to be instantiated in any JVM. Terracotta can load an object from the server automatically as it is needed by an application. When a reference to a clustered object that is not currently instantiated on the local physical heap is traversed, Terracotta will automatically request that objects data from the Terracotta server, instantiate it on the local heap, and wire up the reference that are traversing, transparently from the application point of view.

Clustered POJOs

POJO is an acronym for **Plain Old Java Object**. The name is used to emphasize that a given object is an ordinary Java Object, not a special object, and in particular not an Enterprise JavaBean. POJO refers to a Java object that lives on the heap of the JVM and does not need any extra framework or component to enable the functionality it is intended to provide. Terracotta enables a POJO programming model when running applications in a cluster. The model is represented by the ability to honor the Java Memory Model as well as the language semantics. Specifically, POJO clustering implies that object identity is preserved across JVMs.

Clustered POJOs are the same object in different JVMs, not merely copies or clones that have the same data.

If a POJO object is clustered in the Terracotta virtual heap, it is possible to exploit typical Java object operation available for local heap objects. This include comparisons with the == operator evaluate to true or modifying the fields of a POJO through the normal access (.) operator (e.g., foo.bar = baz) without having to later call some commit or put-back operation. It is also possible to put a POJO Object into a data structure (like a Map) and get it out again without being handed back a copy of the object.

Terracotta guarantees consistency in execution of concurrency operations (e.g., synchronized, wait(), and notify()) on a POJO. To provide this functionality, Terracotta must address cross-JVM **thread communication** issues.

The JVM has simple and elegant built-in mechanisms for coordinating threads: the bytecode instructions **MONITORENTER** and **MONITOREXIT** and the various signatures of the Object methods **wait()** and **notify()**. With Terracotta, thread coordination constructs that exist in the Java language will transparently work across JVM process boundaries. One thread in one JVM can call wait() on an object and another thread in another JVM on another machine can call notify() to wake up the waiting thread. Terracotta exploits **MONITORENTER** and **MONITOREXIT** bytecode instructions as it can be seen in figure 2.8.

Figure 2.8 suggests that Terracotta also needs a cluster-wide lock notion.

Cluster-wide Locks

Terracotta extends the built-in locking semantics of the JVM to have a clusterwide, cross-JVM meaning. Clustered locking is injected into application code based on the locks section of the Terracotta configuration file, each lock configuration stanza uses a regular expression that matches a set of methods.

A lock stanza must be specified as either a **named lock** or an **autolock**. Methods that match an autolock stanza are augmented by Terracotta to acquire a cluster wide lock on a clustered object wherever there is synchronization on that object. Autolock can be seen as an extension of methods existing for Java synchronization to have a cluster wide meaning. Unlike autolocks, named locks function only at method entry and exit. This means that, for

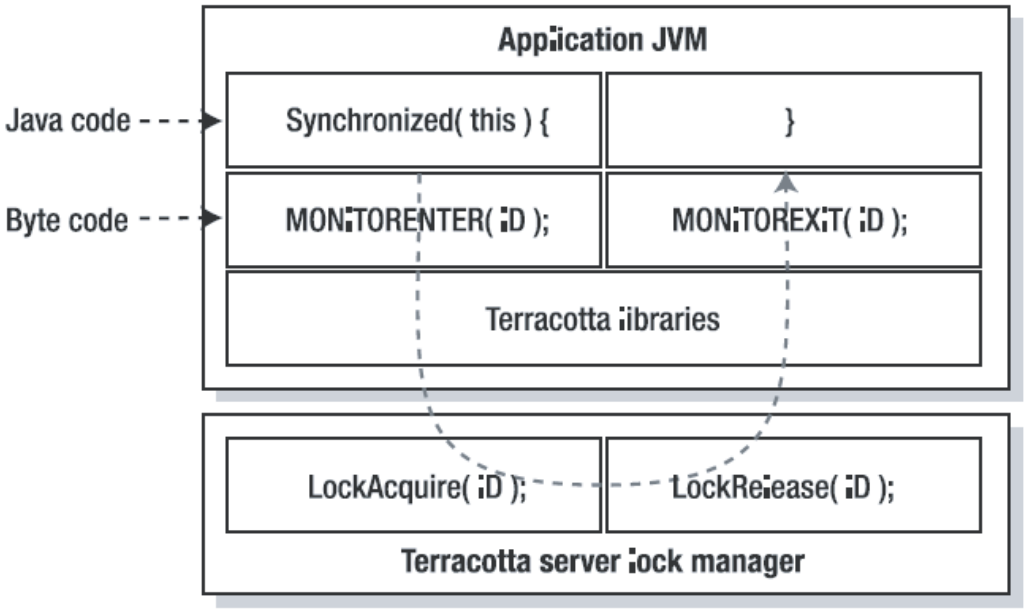


Fig. 2.8. Terracotta hooks into the MONITORENTER and MONITOREXIT bytecodes

methods that match a named lock configuration stanza, a thread must first acquire the lock of that name from the Terracotta server before being allowed to execute that method.

As a result, named locks are very *coarse grained* and should only be used when autolocks are not possible.

Another configuration option in the lock configuration stanza is the lock level. Terracotta locks come in four levels:

- **Write locks** are mutual exclusion locks that act like regular Java locks. They guarantee that only one thread in the entire cluster can acquire that lock at any given time.
- **Synchronous write locks** add the further guarantee that the thread holding the lock will not release the lock until the changes made under the scope of that lock have been fully applied and acknowledged by the server.
- **Read locks** allow multiple reader threads to acquire the lock at a time, but those threads are not allowed to make any changes to clustered objects while holding the read lock.
- **Concurrent locks** are always granted.

2.4 Tools

2.4.1 RePast

RePast [Hohb] is a *free*, *open source* library of classes for creating, running, displaying and collecting data from agent based simulations. In addition, RePast includes several varieties of charts for visualizing data (e.g. histograms and sequence graphs) and can take snapshots of running simulations and create QuickTime movies of such. RePast is a fully Object Oriented architecture created by University of Chicago, Chicago Socials Science Research Computing division.

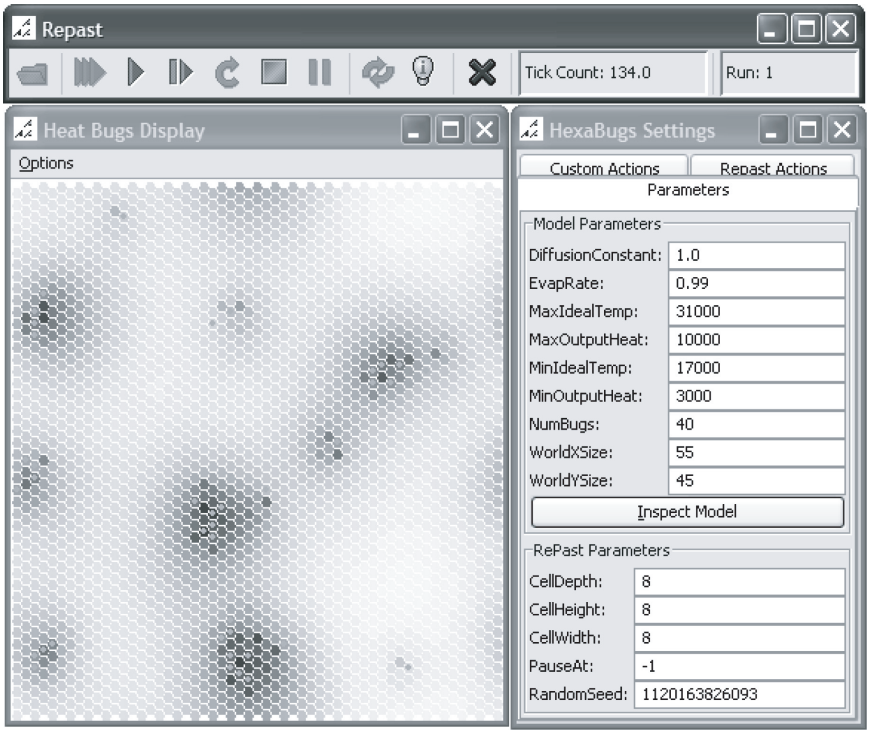


Fig. 2.9. a snapshot of RePast

RePast borrows much from the design of the *Swarm simulation toolkit* [Swa] and can properly be termed a Swarm-like simulation framework.

At its heart, RePast behaves as a discrete event simulator whose quantum unit of time is known as a **tick**. The tick exists only as a hook on which the execution of *events* can be hung, ordering the execution of the events relative to each other. Ticks are merely a way to order the execution of events relative to each other.

In RePast events are referred as **actions**, and a RePast schedule is in charge for correct scheduling/dispatching issues.

RePast simulation is primarily a collection of **agents** of any type and a **model** that sets up and controls the execution of these agents' behaviors according to a schedule. This schedule not only controls the execution of agent behaviors, but also actions within the model itself, such as updating the display, recording data, and so forth. Scheduling can be automated via the model or manually implemented by the modeller. In addition, this model is typically responsible for setting up and controlling simulation visualization, data recording and analysis. The model is said to be composed of these additional components (the schedule, the display, and so forth). There are different implementations of RePast [NCV06]: **RePastJ** for Java, **RepastPy** for python scripting, **Repast.Net** for microsoft .Net framework and the more recent **RePast Symphony** [Hohb]. The following focuses on RePastJ

RePastJ packages

The features offered by RePastJ toolkit are grouped into *packages* that are explained below:

- **Analysis**
  The classes in the analysis package are used to gather, record, and chart data. Using these classes a modeller defines data sources and hooks up recording or charting classes to these sources. Data can be easily recorded in a tabular or customized format and charted in a sequence graph, histogram or user-defined plot.

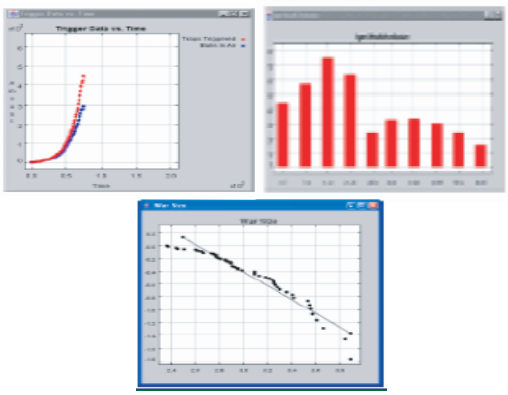


Fig. 2.10. Charting options

- **Engine**
  The engine classes are responsible for setting up, manipulating, and driving a simulation. The *SimModel* interface is the super-class for all models

written with RePast. A partial implementation of SimModel, the *SimModelImp* class is provided and can be used as the base class for most, if not all, models written with RePast. Alternatively, the *SimpleModel* class can be used to automate event scheduling as described above. The controller classes (*BaseController*, *Controller*, and *BatchController*) are responsible for handling user interaction with a simulation either through a GUI or by automating such interaction through the use of a batch parameter file. In addition, the engine package contains the classes that make up the scheduling mechanism.

- **event**
  As mentioned above, the schedule is responsible not only for the execution of agent actions, but also actions within the model itself, such as display updates and so on. However, not all communication between parts of a model is done via the scheduler. A small portion is performed using an event mechanism; these classes together with those in the engine package constitute this event mechanism. These classes are used internally by RePast and are not of real concern to the modeller.
- **Gui**
  The gui classes are responsible for the graphical animated visualization of the simulation as well as providing the capability to take snapshots of the display (see figure 2.11) and make QuickTime movies of the visualization as it evolves over time. The various Display classes work in conjunction with the classes in the space package to display these spaces appropriately. Via a *DisplaySurface*, the *LocalPainter* class handles the actual display of these spaces on the screen, and the DisplaySurface itself allows for the probing of the displayed objects. Probing, left clicking on the visualization of a simulation object, introspects that object (an agent for example) and displays its current parameters in a separate window. The gui package also contains the graph layouts used to visualize networks and an extensible Display class that can be used to build custom displays.
- **Network**
  The network package contains the core classes used to build network simulations. These include default node and edge classes working together such that a node knows its incoming and outgoing edges and an edge knows its source and target node. The *NetworkFactory* class is used to load networks from a file in a variety of formats as well as to generate networks (Small World, Random Density, and Square Lattice). Networks can be recorded as adjacency matrices in a variety of formats using the *NetworkRecorder*. In addition there are some utility classes that can be used to collect some simple yet useful network statistics.
- **Space** In an agent simulation, agents often have some sort of *spatial relationship* to each other. The space package contains classes that instantiate various sort of spacial relationships. The classes themselves are essentially container classes that represent various types of spaces (two-dimensional *grids*, *torii*, single or multiple occupancy, and so forth) accessible through

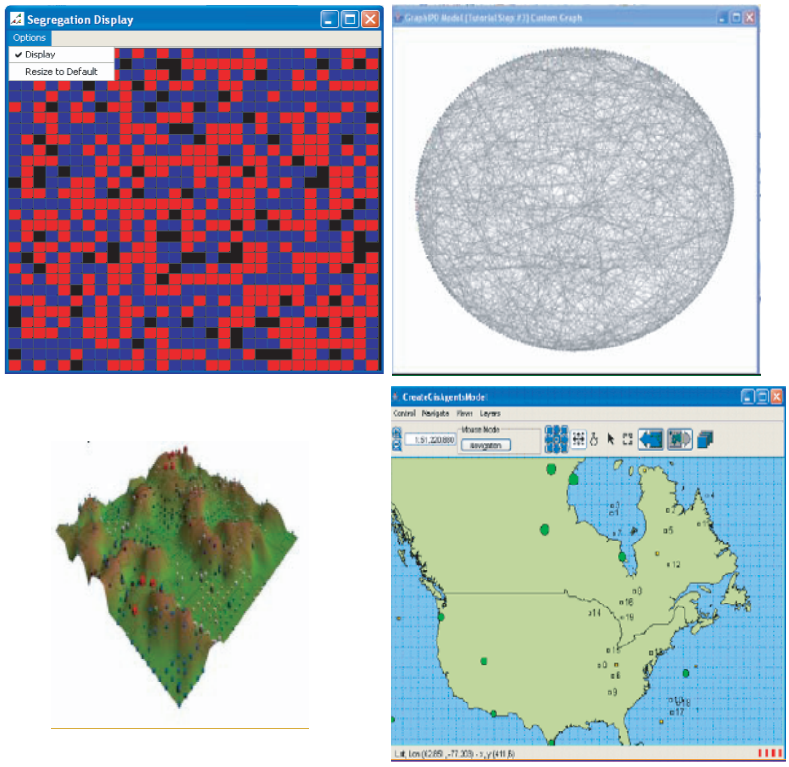


Fig. 2.11. Display example

the appropriate interfaces. For example, the grid spaces allow objects to be inserted and retrieved based on x and y coordinates. Spaces work in conjunction with the display classes in the gui package to present a visualization of the space and the objects (e.g., agents) that it contains.

- **Util**
  This utilities package, contains a variety of utility classes used both internally by RePast and by the modeller. The two most important classes here are *Random* which encapsulates a large number of random number distributions and operations on them, and *SimUtilities* which contains a number of static methods that shuffle lists, display dialogs, update probes and so forth.

2.4.2 Aspect oriented programming: AspectJ

Object-oriented programming (OOP) has been presented in the past as a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. In reality, however, there are many programming problems where OOP techniques are

not sufficient to clearly capture all the important design decisions the program must implement.

There are some programming problems that fit neither the OOP approach nor the procedural approach it replaces. From these considerations comes the so-called **aspect-oriented programming (AOP)**.

Aspect-oriented programming entails breaking down program logic into distinct parts, the so-called **concerns**, i.e. cohesive areas of functionality. All programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g. procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. But some concerns defy these forms of implementation and are called *crosscutting concerns* because they "cut across" multiple abstractions in a program.

Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every single logged part of the system. Logging thereby crosscuts all logged classes and methods.

All AOP implementations have some crosscutting expressions that encapsulate each concern in one place. The difference between implementations lies in the power, safety, and usability of the constructs provided.

AOP provides several tools that can help with crosscutting concern problem. The first is the language used to code the requirements or concerns into units of code (either objects or functions). The AOP literature commonly calls this the *component language*. The secondary or support requirements (aspects) are coded as well, using an aspect language. Nothing in the paradigm states that either language needs to be object-oriented in nature, nor do the two languages need to be the same. The result of the component and aspect languages is a program that handles the execution of the components and the aspects. At some point, the respective programs must be integrated. This integration is called *weaving*, and it can occur at compile, link, run, or load time.

Figure 2.12 shows how a component *Product* is compiled along with the aspect. The compiler weaves the aspect code into the component code to create a functioning system.

AspectJ semantics

AspectJ adds to Java just one new concept, a *join point*, and a few new constructs: *pointcuts*, *advice*, *introduction* and *aspects*.

Pointcuts and advice dynamically affect program flow, and introduction statically affects a program's class heirarchy.

A join point is a well-defined point in the program flow. Pointcuts select certain join points and values at those points. Advice defines code that is executed when a pointcut is reached. These are, then, the dynamic parts of AspectJ.

AspectJ also has a way of affecting a program statically. Introduction is how AspectJ modifies a program's static structure, namely, the members of

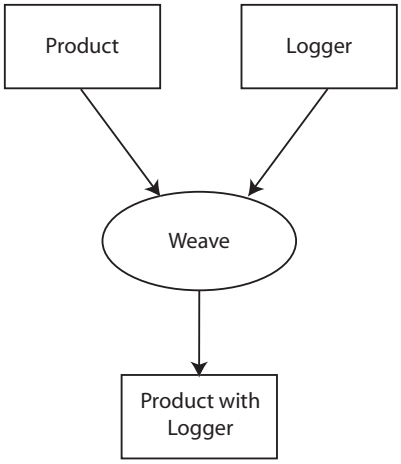


Fig. 2.12. Weaving example

its classes and the relationship between classes. The last new construct in AspectJ is the aspect. Aspects, are AspectJ's unit of modularity for crosscutting concerns. They are defined in terms of pointcuts, advice and introduction.

The Dynamic Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns.

Join points are certain well-defined points in the execution of the program. AspectJ provides for many kinds of join points, for example: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including normal or abrupt return. Each method call itself is one join point. The dynamic context of a method call may include many other join points: all the join points that occur when executing the called method and any methods that it calls.

Pointcut Designators

In AspectJ, pointcut designators (or simply pointcuts) identify certain join points in the program flow. For example, the pointcut

```
call(void Point.setX(int))
```

identifies any call to the method *setX* defined on *Point* objects. Pointcuts can be composed using a filter composition semantics, so for example:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

identifies any call to either the *setX* or *setY* methods defined by *Point*. Programmers can define their own pointcuts, and pointcuts can identify join points from many different classes, in other words, they can crosscut classes. So, for example, the following declares a new, named pointcut:

```
pointcut move(): call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int)) ||
call(void Point.setY(int)) ||
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point));
```

The effect of this declaration is that *move* is now a pointcut that identifies any call to methods that move figure elements.

Property-Based Primitive Pointcuts

The previous pointcuts are all based on explicit enumeration of a set of method signatures. This is called *name-based crosscutting*. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. This is called *property-based crosscutting*. The simplest of these involve using wildcards in certain fields of the method signature. For example:

```
call(void Figure.make*(..))
```

identifies calls to any method defined on *Figure*, for which the name begins with *make*, e.g. *makePoint*, *makeLine* and so on;

```
call(public * Figure.* (..))
```

identifies calls to any public method defined on *Figure*. One very powerful primitive pointcut, *cflow*, identifies join points based on whether they occur in the dynamic context of another pointcut. For example:

```
cflow(move())
```

identifies all join points that occur between receiving method calls for the methods in *move* and returning from those calls (either normally or by throwing an exception)

Advice

Pointcuts are used in the definition of **advice**. AspectJ has several different kinds of advice that define additional code that should run at join points.

- *Before* advice runs when a join point is reached and before the computation proceeds, i.e. it runs when computation reaches the method call and before the actual method starts running.
- *After* advice runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller.
- *Around* advice runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all.

Introduction

Introduction is AspectJ's form for modifying classes and interface and their hierarchy. Introduction adds new members to classes and alters the inheritance relationship between classes. Unlike advice that operates primarily dynamically, introduction operates statically, at compile time. Introduction changes the declaration of classes, and it is these changed classes that are inherited, extended or instantiated by the rest of the program.

Consider the problem of adding a new capability to some existing classes that are already part of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to each affected class a method that implements this interface. AspectJ can do better. The new capability is a crosscutting concern because it affects multiple classes. Using AspectJ's introduction form, we can introduce into existing classes the methods or fields that are necessary to implement the new capability.

Aspect Declarations

An aspect is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers. The crosscutting implementation is provided in terms of pointcuts, advice and introductions. Only aspects may include advice, so while AspectJ may define crosscutting effects, the declaration of those effects is localized.

3

The Theatre architecture

3.1 Introduction

Agents are characterised by their computational autonomy, mobility, proactivity and communication capabilities. To handle scalability and complexity of particular systems to be analyzed, e.g. large-scale dynamic structure systems, telecommunications, business process modelling, computer games etc., parallel and distributed simulations are often required [JA06, LT01, MT08, LLT07, CFNP07]. In an open multi-agent system, agents can dynamically join or abandon the simulation model. Open multi-agent distributed simulation systems challenge for proper agent naming solutions and efficient communication mechanisms despite dynamic migration [VA01b, JAA05, WVP+05].

In the work described in this chapter the actor model of computation [Hew77b, Agh86] is assumed as the starting point for building multi-agent systems. Concrete multi-agent systems directly founded on the actor vision include ActorFoundry [Ast99] and SALSA [VA01b]. SALSA adopts a naming mechanism for actors which influenced the solution described in this chapter.

The Theatre architecture is based on Java and actors. It differs from Agha's actor model in that it uses a light-weight notion of actors. Theatre actors are reactive thread-less objects instead of being self-active objects, i.e. equipped of internal threads of control. Reactive actors favor timing predictability in real-time applications [NP01], facilitate the expression of custom message-based scheduling/dispatching strategies in execution theatres, simplify migration operations and make it possible to build large-scale, high-performance and dynamically reconfigurable distributed applications [FNP02, CFN06, CFN07b, CFNP07].

Theatre naturally supports variable structure systems through agent-migration and location-transparent naming. It embodies mechanisms as in [JAA05] for efficient agent communications. In addition, Theatre can work with strategies for load-balancing as described in [JA06]. This particular actor model was also successfully employed in a time-warp algorithm

[Fuj00, BNO03] e.g. devoted to the distributed simulation of modular time Petri nets [CFN07a].

In order to deploy Theatre as an architecture for parallel/ditributed simulation, *time management* and *transport layer* are realized through standard middleware thus opening to simulator interoperability and reuse. Two different middleware which can be adopted are considered: **HLA/RTI** (see 2.3.1) for execution e.g. upon a computer network and **Terracotta** (see chapter 5) for parallel execution e.g. on a *multi-core* hardware architecture. Other solutions exist (e.g. based on Java sockets and Java RMI).

3.2 Theatre basics

The architecture of Theatre consists of two main parts:

- *execution platforms*, i.e. theatres, which provide "in-the-large" features, i.e. the environmental services supporting actor execution, migration and interactions. Services are made available to actors through suitable APIs
- *actor components*, i.e. the basic building blocks "in-the-small", which capture the application logic. In this chapter the terms actor and agent will be used interchangeably.

Theatre can be hosted by different object-oriented programming languages. In this work, though, Java is used as the implementation language (see also [CFNP07]).

3.2.1 Actor modeling and behavior

Actors are reactive objects which encapsulate a data state and communicate to one another by asynchronous message passing. Messages are typed objects. Actors are at rest until a message arrives. Message processing is atomic: it cannot be suspended nor preempted. Message processing represents the unit of scheduling and dispatching for a theatre. The dynamic behavior of an actor is modelled in a basic case as a finite state machine (see Fig. 3.1) that is synthesized in the `handler(message)` method which receives the message to process as a parameter. Responding to a message causes in general the following actions to be executed:

- new actors are (possibly) created
- some messages are sent to known actors (*acquaintances*). For proactive behavior, an actor can send to itself one or more messages
- the actor migrates to another theatre
- the message can be deferred by remembering it in state or data
- current state of the actor is changed (become operation).

User-defined actor classes are heir of the Actor base class. Message classes are heir of Message base class. Message deferral can be required when the received message cannot be processed and answered by the agent in its current state. Each actor is without internal threads. Therefore, message handling extends the control thread of the theatre within which the agent runs.

It is worth noting that the use of actors with an internal thread of control as in [JA06] makes it difficult to achieve an efficient and scalable simulation framework. Explicit messages must be exchanged with the control engine to avoid virtual time inconsistencies among actors. In addition, message delivery suffers from space/time overhead caused by reflective method calls and thread-level context-switches. In the actor model adopted in this paper, though, delivering a message to a local actor costs a normal method invocation and no context-switch. Moreover, when a message processing terminates, the engine control loop is automatically and implicitly re-entered.

Being thread-less, a huge number of application actors can be created with very limited demand on the underlying operating system resources and services with respect to thread-based agencies. All of this improves model scalability while ensuring the achievement of good execution performance [CFN07b] (see also later in this thesis).

The actor concept can be related to that of "Logical Process" (LP) which is well-known in the parallel/distributed simulation field [Fuj00]. Different design scenarios can be considered. For example, in [BNO03, CFN07a] a single coarse-grain actor is used as an LP allocated for execution on a physical processor/theatre, and managed by time warp operation (state saving and restoring). In this paper, as in [CFN07b], an LP consists of a (dynamic) collection of fine-grain actors, operated under a conservative simulation control structure.

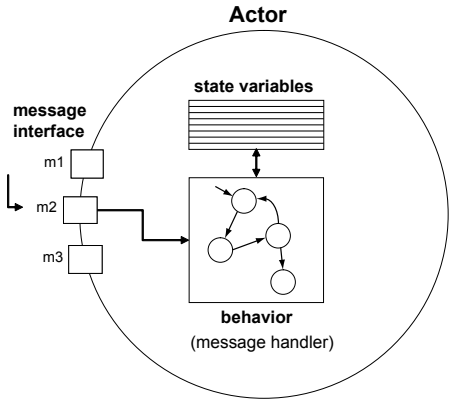


Fig. 3.1. Actor structure

3.2.2 Structure of a theatre

Basic components of a theatre are (see Fig. 3.2):

- an instance of the Java Virtual Machine (JVM)
- a Control Machine (CM)
- a Transport Layer (TL)
- a Local Actor Table (LAT)
- a Network Class Loader (NCL).

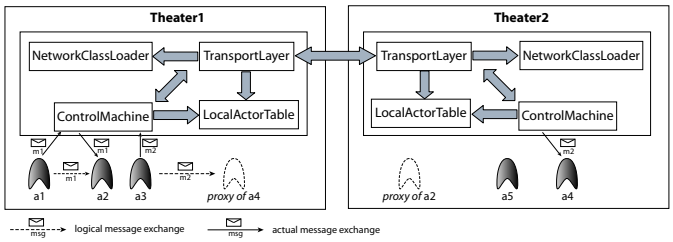


Fig. 3.2. Theatre architecture

The Control Machine hosts the runtime executive of the theatre, i.e. it offers basic services of message scheduling/dispatching which regulate local actors. The Control Machine can be made time-sensitive by managing a time notion ("real" or simulated). Actually, the Control Machine organizes all pending (i.e., scheduled) messages in one or multiple message queues. Instead of having one mail queue per actor [Agh86], the Control Machine buffers all sent messages and superimposes to them an application-dependent control strategy. During the basic control loop, a pending message is selected (e.g., the most imminent in time) and dispatched to its destination agent by activating the relevant `handler()` method. At the `handler()` termination, the control loop is re-entered, it schedules all messages sent by the last activated agent and, finally, starts its next cycle. The Transport Layer furnishes the services for sending/receiving network messages and migrating agents. The Local Actor Table contains references to local agents of the theatre. The Network Class Loader is responsible for getting dynamically and automatically the class of an object (e.g., a migrated agent) by retrieving it from a network Code Server acting as a code repository for the distributed application.

3.2.3 Agent naming

At its creation, an actor receives its unique id (string) and its Java reference gets included in the Local Actor Table of the creating theatre, with the actor-id used as a key. After that, the agent can be migrated to a different theatre

and so forth. The Java reference of an agent persists despite migration. After migration, in the Local Actor Table of the source theatre the agent reference is kept but now refers to a *shadow version* (proxy) of the agent, which behaves as a *forwarder*. The forwarder keeps the network information about the destination theatre where the agent migrated. Dispatching a message to a forwarder automatically generates a network message toward the destination theatre. In general, a certain number of hops can be required to reach the agent. Of course, during its migration, an actual agent can come back to a theatre where a forwarder of it exists. In this case, the shadow agent gets its state replaced with that of the real agent. Agent migration implies the relation proxy/normal of its acquaintances to be reviewed according to the viewpoint of the receiving theatre. Some acquaintances become proxies because the corresponding actors reside in a remote theatre, e.g. that from which the mobile agent originated. Other acquaintance references can change from proxy to normal in the case the referred actor is local to the migrated site. The update operation is accomplished with the help of the Local Actor Table and by using location information which is carried by the migrated actor.

The universal actor name [VA01b] of an agent consists of the tuple `<theatre-id, actor-id>`. The `theatre-id` is the tuple `<host-name, port>` where `host-name` is the IP address of the computing node where the theatre is allocated, and `port` is a unique port number associated with the theatre. To improve the efficiency of communications, a network message actually counts the number of hops performed to reach its destination and automatically asks for an update of the addressing information in the forwarder agent in the source theatre with the actual position of the destination agent.

3.2.4 Agent migration

A customization of the standard Java serialization mechanism is used to support the transmission of actors and messages between theatres. To comply with proxy/normal representations of actors and the persistence of local actor references despite migrations, actor parameters in messages and acquaintances in actors are transparently replaced, during serialization, with serialized objects of an auxiliary class ActorInfo. An actor info carries the following information about an agent: actor name, class name, IP address and port of residing theatre. To enable replacement, the Actor base class redefines the default `writeReplace()` method which is able to distinguish if an actor is under migration or if it is part of a message/actor which is migrated, in which case its standard serialization is replaced.

When a serialized message/actor object is received by a theatre it gets de-serialized in the standard way. A serialized actor info object, though, is handled by invoking on it the default `readResolve()` method which is redefined by the ActorInfo class. It is responsibility of `readResolve()` to adjust an actor parameter or acquaintance to the reference to a newly created or already

existing proxy actor or to an already existing normal actor. The adjust operation relies on the information in the Local Actor Table. In the case a normal actor finds a proxy version of itself in the receiving theatre, the proxy status gets updated with that of the normal actor with the help of object introspection as permitted by the features of the Java API `java.lang.reflect`. Agent mobility follows a *weak migration* semantics [BHRS98]. The absence of internal threads in an actor reduces the migration to the serialization/deserialization of the actor data status. No thread control status has to be saved and subsequently restored. A migration request is implemented with the help of a behind of the scene Migrate message. Therefore, migration is not synchronously executed as part of a message handling.

3.2.5 Dynamic model reconfiguration

Theatre specifically addresses modelling and simulation of complex systems which are component-based [CFNP07], timed, mobile and whose structure can change during runtime [HZM05, JRT+03, JA06, PV07]. Such systems are not adequately supported by conventional M&S tools where the structure is often assumed to be static and dynamism only relates to state changes caused by the occurrence of events. However, many systems exist (e.g. predator/prey models in biology, adaptive networks in telecommunication systems accommodating for the presence of mobile users, and so forth) which require structure dynamism for them to be effectively modelled and analyzed.

As in Kiltera [PV07] model adaptivity depends on *link mobility*, i.e. the possibility of reconfiguring during runtime the interconnection infrastructure of agents by adjusting the acquaintance network of the system. The approach preserves the contract of agents' message interfaces and it is very flexible when paired with the mobile capabilities of actors which can migrate among the theatres allocated to different physical nodes of a distributed system.

3.3 Theatre on top of HLA[1]

HLA (see 2.3.1) was chosen because it is a standard middleware for distributed simulations, offers time management services which help structuring the simulation control engine, favours interoperability with existing (possibly heterogeneous) HLA-compliant simulators, thus promoting simulation reuse. However, distributed simulation engines for Theatre were also implemented outside of HLA using such transport layers as Java Socket and Java RMI [CFN07b, CFN07a].

A key point of the approach is a fine-grain migration process compliant with HLA, which occurs at the agent level and not at the federate level [TPA05, CYLT05]. The migration mechanism is transparent to HLA. It does

[1] based on [DAAL07]

not depend on federation wide synchronization based on save/restore operations. Moreover, migration does not introduce "freeze" constraints on the federation.

Theatre makes mobile agents fully integrated in an HLA distributed simulation. With respect to similar frameworks, e.g. Cougaar [KFC04], Theatre distinguishes for its simplicity and lean implementation. In other approaches, e.g. [LHF05], agents are often used for supporting data-filtering strategies from an HLA based simulation.

The proposed UAV model is highly dynamic and introduces communication patterns among mobile agents (UAVs and targets) which are determined at runtime. The goal is to experiment with different engagement and attack strategies. The example inherits from the *agent-environment model* [WVP+05, LT01] in the sense that agents can interact indirectly with other agents by communicating with the environment (itself abstracted as an agent). This solution can improve agent communications and the performance of simulations.

Theatres naturally map on to HLA federates (see Fig. 3.3). The translation, in particular, was designed so as to keep the Java-based agent programming level totally unaware of HLA, with the Control Machine component of a theatre which hides all the machinery required for interfacing RTI. The prototyped solution basically uses HLA as a transport layer and exploits its time management services.

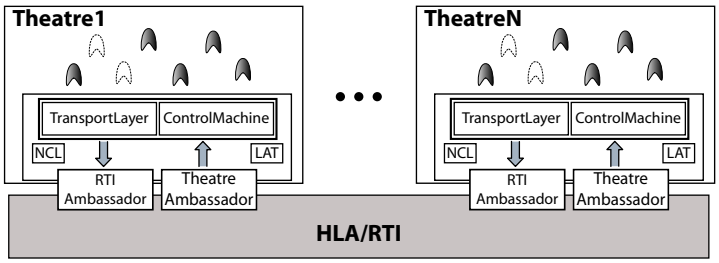


Fig. 3.3. Theatre over HLA

The event-driven character of actors, which depends on explicit message passing, suggested the use of only *interaction classes* in order to ensure communications between the federates of a Theatre federation. A specific interaction class is responsible for carrying all the message/actor exchanges between an ordered pair of federates. As a consequence, in the Federation Object Model (FOM) of a federation with N theatres, a collection of $N \times (N-1)$ interaction classes should be anticipated. A generic interaction class takes as a parameter a byte array which can store a serialized message or actor object. At configuration time, a table is initialized in each Control Machine which maps a partner `theatre-id` with the corresponding interaction class. A theatre t publishes

all the interaction classes which have t as source (output interaction classes) and subscribes to all the interaction classes which have t as destination (input interaction classes).

3.3.1 Time management

The following considers an event-driven conservative Control Machine of a *time-constrained* and *time-regulating* theatre/federate. Except for the initialization phase, a distributed simulation is based on the exchange of time stamped messages, i.e. an interaction object is always provided of its occurrence time stamp. The Control Machine manages the federate logical time and a Timed Message Queue (TMQ) which holds, ranked by ascending time stamps, both local and external received simulation messages.

Fig. 3.4 shows in pseudo-code the operation of the control loop. Provided the TMQ is not empty, the control engine asks RTI to advance the local logical time to the minimal time stamp in TMQ. While the control loop is waiting for the grant, RTI can deliver to the theatre a set of external originated messages whose time stamp (effective granted time) is less than or equal to the proposed time advancement. Each such message gets scheduled on the TMQ. Following a grant, the logical time is advanced to the time stamp of the next imminent message in TMQ which is then dispatched to its destination actor. Finalization messages, i.e. messages destined to transmit to a supervisor agent the simulation results collected by the federate, scheduled with the simulation time limit tEnd as time stamp, can be pre-allocated in the TMQ. As a consequence, when the simulation time limit is reached, finalization messages get executed in the normal way.

When TMQ is empty, the federate asks RTI to advance its logical time to tEnd+1. On the arrival of the corresponding grant, the control loop is exited. After that, the federate typically resigns from the federation. In the worst case, a Theatre federation can require to execute with zero lookahead.

3.3.2 Lifecycle of a Theatre-based mederation

Execution of a Theatre federation requires preliminarily the files FED, containing the FOM, and TheatreConfig.properties, containing configuration data, to be prepared. TheatreConfig.properties specifies such information as the number of participating federates and the names of the interaction classes. An additional Manager federate is assumed in the federation, whose goal is to properly boot and terminate federation execution. Manager is not time-constrained nor time-regulating. At start-up, each federate tries to create the federation. After that every federate joins the federation. Following its own join, the Manager federate waits for all the remaining federates to join the federation. Then the Manager creates a *synchronization point* ("Start") which RTI announces to all the participating federates. The purpose of the synchronization point is to ensure that no federate starts its control loop and then begins to exchange time

```
ask RTI to advance logical time to 0
wait for grant
logical time=0
loop
    if( TMQ is not empty ){
        let minTS be the time stamp of the first message in TMQ
        if( minTS > logical time ){
            ask RTI to advance logical time to minTS
            wait for grant
            let minTS be the time stamp of the first message in TMQ
            logical time = minTS
        }
        extract the first message in TMQ and dispatch it
            to its destination actor
    }
    else{//TMQ is empty
        ask RTI to advance logical time to tEnd+1
        wait for grant
        if( grant arrived for advancement to tEnd+1 ) break
    }
end-loop
```

Fig. 3.4. Control loop of a theatre/federate

stamped messages, while some federates still exist which are not completely initialized. The initialization actions of a federate are summarized in Fig. 3.5.

```
ask RTI and wait grant for becoming time-constrained
ask RTI and wait grant for becoming time-regulating
publish output interaction classes
subscribe input interaction classes
achieve the Start synchronization point
start the control loop of the simulation engine
```

Fig. 3.5. Federate initialization work

Manager gets notified each time an actor federate resigns from federation. When there are no other federates except itself, Manager resigns and then destroys the federation.

3.3.3 An UAV modeling and simulation example

As a test bed of using Theatre over HLA/RTI, a complex simulation model was developed which is highly dynamic and based on mobile agents whose identity and communication patterns are discovered at runtime. The model is original and is concerned with coordination and control of Unmanned Aerial Vehicles (UAVs) [JRT+03, JA06] devoted to shooting moving targets over a territory of interest (mission area) in the presence of *static* obstacles. The goal is to study and compare different engagement strategies of UAVs with respect to targets.

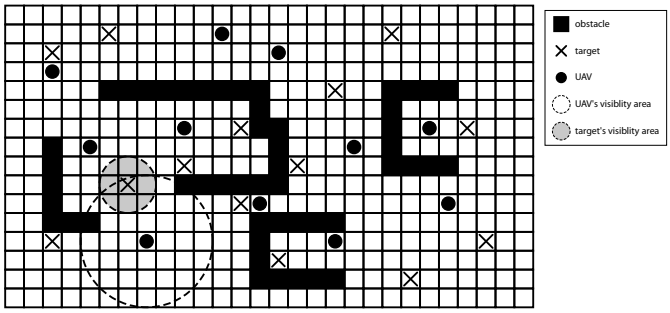


Fig. 3.6. UAV mission area

UAVs are aircrafts that operate without needing human control and have an autonomous behavior in accomplishing their mission. An UAV does not know in advance the target locations nor the topology of the territory. After reaching the mission area, an UAV starts the exploration by flying low in order to discover the presence of interesting entities in its neighborhood with the help of its radar. The information about number and positions of surrounding entities influence UAV behavior. Each UAV is supplied with a certain amount of fuel and ammunition. UAVs begin exploration always starting from a certain zone (entry area). An UAV can operate in the mission area while it has enough fuel and ammunition. When it needs to stock up on something, an UAV starts to fly high to reach its base (located outside the mission area) and after comes back in the entry area to continue its mission. During an engagement, an UAV tries to hit the target in order to consume target energy. Targets move over the territory and, like UAVs but with a small visibility range, can detect the presence of obstacles or other entities.

The mission area is modeled as a bi-dimensional grid (see Fig. 3.6) where each cell is supposed to be large enough to host up to a certain number of UAVs and targets. An obstacle is supposed to occupy a whole cell thus excluding other entities in the same cell. Information about (part of) the mission area is handled by a specific environment actor (EnvActor). UAVs and targets are modeled as actors (UAVActor and TargetActor) whose behaviors dictate how the respective entities dynamically reacts to the information about the surrounding environment. This information is obtained by exchanging messages with the EnvActor.

UAV Behavior

The behavior of an UAVActor is modeled by the finite state automaton depicted in Fig. 3.7. At its start-up, the UAVActor finds itself in the INIT state. When the actor receives an Init message, which contains initial configuration data such as the UAV position in the mission area, the engagement strategy (see section 3.3.3) and the storage of fuel and ammunition, it switches to the

EXPLORE state. UAVActor waits there for the arrival of an EnvInfo message, coming from the local EnvActor, that brings information about the UAV visible surrounding environment. This mirrors the information-retrival process of a real UAV which is based on its onboard radar system.

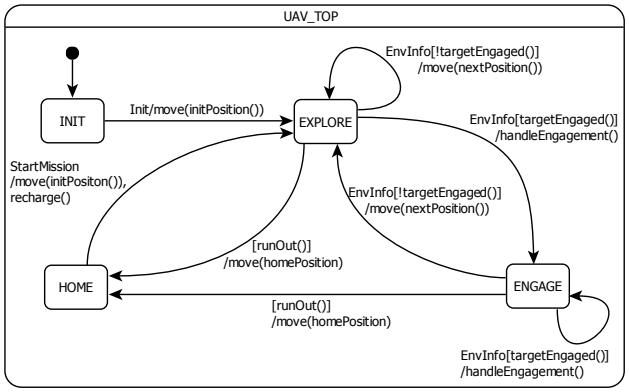


Fig. 3.7. UAVActor behavior

In the case in the current cell there are no targets to engage, the UAVActor stays in the EXPLORE state and continues exploring the mission area. It moves to one of its surrounding cells by first computing the direction to follow and then sending a timed message communicating its next position to the EnvActor (see also Fig. 3.11). The EnvActor will reply with an EnvInfo message at the time the UAVActor reaches its destination cell. When the UAV moves from a cell to another, it consumes its fuel in proportion to the covered distance.

If the UAVActor detects in the current cell the presence of an *interesting* target (e.g. one which is not already engaged by another UAV), it switches to the ENGAGE state where it begins the engagement by interacting with the corresponding TargetActor. The interaction is actually carried out by having the UAVActor which sends an Hit message to the TargetActor. After having shooted the target, the UAVActor remains in or abandons the ENGAGE state depending on the adopted strategy (see section 3.3.3) and on the position and state of the visible targets. In particular, a target may disappear because it has been destroyed or has become hidden behind an obstacle. In these cases the UAVActor goes back to the EXPLORE state where it resumes exploration.

An UAV will eventually run out of fuel or ammunition thus requiring to come back to the base station. UAVActor accounts for these situations by the two transitions leaving EXPLORE and ENGAGE states and reaching the HOME state.

While an UAV moves over the mission area, the direction to follow is computed according to a discrete random variable θ.

Let $Dir = \{N, NE, E, SE, S, SW, W, NW\}$ be the set of directions, θ is defined as follows:

$$\theta(N) = 0, \theta(NE) = \frac{\pi}{4}, \ldots, \theta(NW) = \frac{7}{4}\pi$$

Let $w_{uav} : Dir \mapsto \mathbb{R}^*$ be the *weight* function that maps each direction d to a non-negative real number. The mass function of θ is defined as:

$$f_{\theta(d)} = \frac{w_{uav}(d)}{\sum_{d' \in Dir} w_{uav}(d')}$$

The values of w_{uav} are calculated from the following quantities.

Directional Term. In the absence of interesting entities, an UAV explores the mission area. In this case, the choice of the direction to follow is determined by the following directional term, which favors the direction taken at the previous step by giving it the maximum weight:

$$w_{dir}(d) = C_{dir} \cdot e^{-\frac{\delta(d,d_p)^2}{k}}$$

$$\delta(d_1, d_2) = \begin{cases} |\theta(d_1) - \theta(d_2)| & \text{if } |\theta(d_1) - \theta(d_2)| < \pi \\ 2\pi - |\theta(d_1) - \theta(d_2)| & \text{otherwise} \end{cases}$$

where d_p is the direction previously followed by the UAV, C_{dir} is the maximal value, which is assumed for $d = d_p$, and k is a parameter that accounts for the influence of the deviation from d_p. Function δ gives the value of the minimal angle between the two directions.

Attractive terms. Let t be a target that is inside the visibility radius of the UAV and that is not obscured by any obstacle. The attractive term due to t is computed as:

$$w_t(d) = T_{max} \cdot e^{-\alpha \cdot \Delta(t)} \cdot e^{-\frac{\delta(d,d_t)^2}{k}}$$

where $\Delta(t)$ is the distance between t and the UAV, d_t is the direction of minimal distance from the target, α is a decay factor which controls how this term decreases with the distance, T_{max} is the maximum value that can be assumed by such term and k has the same meaning as in the directional term.

Obstacle repulsive factor. Each obstacle, which is located in the immediate surroundings of the UAV, must be avoided. As a consequence the direction bringing to the obstacle must not be considered and then the corresponding value of the weight function must be zero. This is accounted by the obstacle repulsive factor $w_{ro}(d)$ which assumes value zero if there is an obstacle located in the adjacent cell along direction d, and value 1 otherwise.

UAV repulsive factor. In order to avoid collisions among UAVs there is a threshold on the maximum value of UAVs which can stay together in the

same cell. The UAV repulsive factor avoids the directions which, if taken, may lead to a violation of this property and it is defined as follows:

$$w_{ru} = \begin{cases} 0 \text{ if } nr(d) \geq MaxUAV \\ 1 \text{ otherwise} \end{cases}$$

where $nr(d)$ is the total number of UAVs located in the cell which would be reached by taking direction d and in its eight adjacent cells, and $MaxUAV$ is the threshold value. The weight function is then computed as follows:

$$w_{uav}(d) = \left(w_{dir}(d) + \sum_{t \in T} w_t(d) \right) \cdot w_{ro}(d) \cdot w_{ru}(d)$$

where T is the set of targets that are visible by the UAV. If there are no entities in the visibility radius of the UAV, the set T is empty, and both repulsive factors are equal to 1. The decision of the next direction is then determined only by the directional term. If T is not empty, the influence of attractive terms should prevail over the directional term. This is obtained by choosing $T_{max} \gg C_{dir}$.

Engagement Strategies

Similarly to [JRT+03] three different engagement/shooting strategies for UAVs were considered.

- **First strategy** (STR1). After it has fired one shoot, the UAV stays in the same cell and following a little while tries to shoot again. If in meanwhile there are no more targets in the cell (because they left or are destroyed) the UAV starts again its exploration
- **Second strategy** (STR2). Once a target has been engaged, the UAV remembers target identity. In the case the target moves, while remaining visible, the UAV tries to follow it and ignores other targets. If the UAV looses visibility of the engaged target, the latter is no longer considered as engaged
- **Third strategy** (STR3). It is similar to STR2 but in this case targets already engaged are not interesting for exploring UAVs.

The goal of STR2 and STR3 is to maintain the UAV focus only on the engaged target. In addition, STR3 avoids that multiple UAVs insist on the same target.

Target Behavior

Fig. 3.8 illustrates the automaton modeling the behavior of TargetActor. A target moves on the mission area trying to escape from the UAVs it detectes in its surroundings. The way a target moves is in part similar to that of an

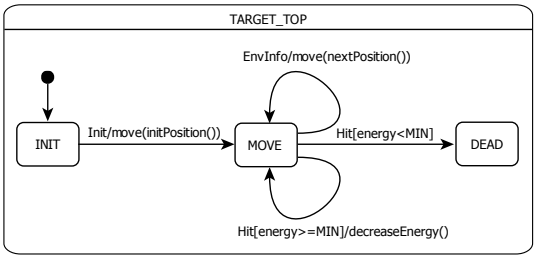


Fig. 3.8. TargetActor behavior

UAV. At each step it chooses the direction to follow by using the same random variable θ but the weight function is now the following:

$$w_{target}(d) = \left(w_{dir}(d) + \sum_{u \in U} w_u(d) \right) \cdot w_{ro}(d) \cdot w_{rt}(d)$$

where, the directional term and the obstacle repulsive factor are computed in the same way as in w_{uav}, w_{rt} is a target repulsive factor which is analogous to w_{ru}, U is the set of UAVs which are visible by the target, and w_u is an UAV repulsive term. w_u is defined as follows:

$$w_u(d) = U_{max} \cdot e^{-\beta \cdot \Delta(t)} \cdot e^{-\frac{\delta(d, rev(d_u))^2}{k}}$$

where $\Delta(t)$ is the distance between the UAV u and the target, d_u is the direction of minimal distance from u, β is a decay factor which controls how this term decreases with the distance, U_{max} is the maximum value that can be assumed by such term, k has the same meaning as before and $rev : Dir \mapsto Dir$ is a function that gives the direction which is opposite to its argument, thus mirroring the goal of a target which is to escape from UAVs.

EnvActor Behavior

In the context of a distributed version of the simulation model, the mission area is partitioned among more theatres. For this reason, an actor modeling a moving entity needs to migrate from one theatre to another when its next position is located outside the portion managed by the current hosting theatre. In addition, when a moving actor is located in the nearness of the local boundaries, it needs also information about the entities inside its visibility radius but located in a different partition. Both issues are transparently addressed by EnvActors (see Fig. 3.9 and Fig. 3.10).

For each theatre there is one EnvActor whose role is handling the environmental information [WPM+05] that may be of interest for the UAVActors and/or TargetActors located inside the theatre. In addition to the local portion of the mission area, the EnvActor maintains an updated snapshot of the

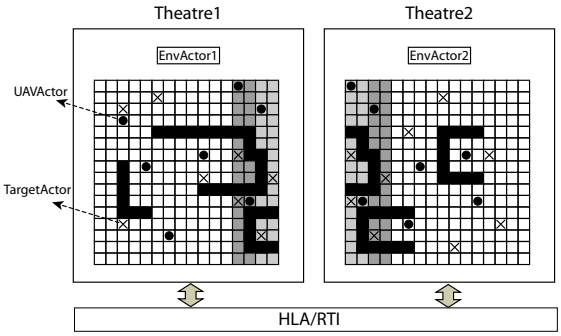


Fig. 3.9. Model partitioning

environment parts that are not handled locally but which may fall in the visibility radius of a local moving entity. Fig. 3.10 portrays boundary details among adjacent (sub) environments. The EnvActor exchanges messages with its neighbor peers (see Fig. 3.11) by sending them updates every time a local event causes a change in a shared zone of interest handled locally, and by receiving updates from the other EnvActors notifying a change in a shared zone handled outside.

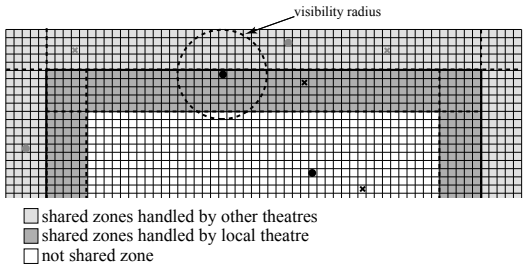


Fig. 3.10. Boundary cells among adjacent (sub) environments

When a TargetActor or an UAVActor wants to move, it sends to the EnvActor a timed message communicating its next position. The time stamp of the messages mirrors the time needed to reach the destination cell. If this position is of competence of the EnvActor, the EnvActor replies with a message containing information about the area in the visibility radius of the sender. Otherwise the EnvActor migrates the sender to the relevant theatre and asks the pertinent EnvActor to reply to the migrating actor. Each moving actor updates its local reference to the EnvActor every time it receives a reply message. On the basis of the information contained in this message the actor regulates its next decision. The described protocol makes TargetActors and UAVActors unaware of distributed simulation concerns.

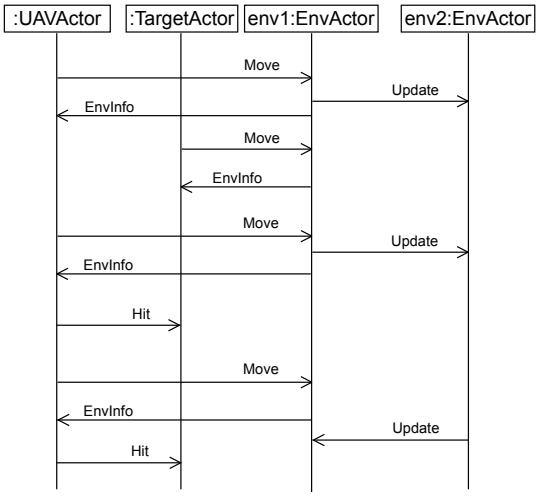


Fig. 3.11. A typical interaction scenario

Model Bootstrap

The mission area is configured by means of an editor application (Fig. 3.12) that allows the visual specification of the territory layout, i.e. the obstacle positions and the initial placements of targets and UAVs. Targets are represented by × marks and UAVs by black circles. Remaining symbols denote obstacles, e.g. walls. The territory configuration is saved on a file, which is fed to the simulator. After federation startup, the distributed model gets actualized by creation, on a predefined theatre, of a Master actor which is in charge of making and deploying the application actors over the theatres according to the model specification. The Master actor migrates on each theatre and creates there the instance of the local EnvActor. After all the EnvActors have been established, the Master actor sends to each of them an Init message containing the specification of the part of the mission area assigned to that theatre. The Master actor tries to split equally the territory among the theatres. Each EnvActor handles the Init message by creating its own representation of the territory, populating it and finally replying to the Master. When all the reply messages are received, the Master broadcasts a Start message to the EnvActors. At this point, model initialization is completed and the simulation actually begins. The Master actor is also in charge of collecting simulation results at the end of the simulation.

Simulation Scenario

A first simulation scenario was chosen for comparing the engagement strategies (see section 3.3.3). The simulation model consists in the grid territory

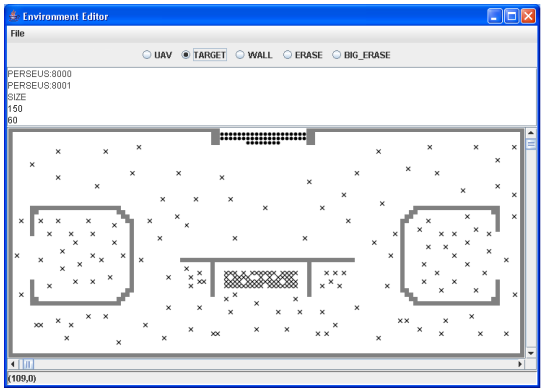


Fig. 3.12. Editor pane used for configuring a territory with targets, UAVs and obstacles

shown in Fig. 3.12, which has dimension of 150×60 cells. A total number of 200 targets were deployed over the mission area and a number of UAVs ranging from 20 to 56 were used. The placement of obstacles was chosen in order to investigate the exploration capabilities according to moving strategies. Table 3.1 collects the values of the parameters adopted for the simulation experiments (s.u., t.u., f.u. and e.u. respectively denote space, time, fuel and energy units). Each time a target is hit it loses 10% of its initial energy. For the sake of simplicity it was assumed that targets have no fuel limitations. When an UAV needs to stock up on something it employs 50 t.u. in order to be again in the mission area.

3.3.4 Experimental mesults

The simulation model was separately configured, validated and executed on a single machine and through a federation of three theatres plus the manager federate, using pRTI1516 [Pit] as HLA implementation. The three theatres, each of which is assigned a distinct region of 50×60 cells of the entire territory, and the manager were allocated on three Pentium IV, 3.4GHz, 1GB RAM, WinXP platforms interconnected by a Gigabit Ethernet switch. The goal of the experiments was to evaluate the performance of the various engagement strategies with respect to the time needed to complete the mission, i.e. to achieve that all targets are consumed. Fig. 3.13 shows a snapshot of the viewer, useful for monitoring the evolution of the model for debugging purposes.

During the effective simulation work the viewer is disabled. All the simulation results were obtained as the mean value of five runs. Fig. 3.14 illustrates how the mission time varies in function of the number of UAVs whilst Fig. 3.15 shows the corresponding fuel consumption. As one can see, strategies STR2 and STR3 perform better than STR1 in any case. In addition, STR3 gives slight better results than STR2. This is due to the fact that in STR1 an

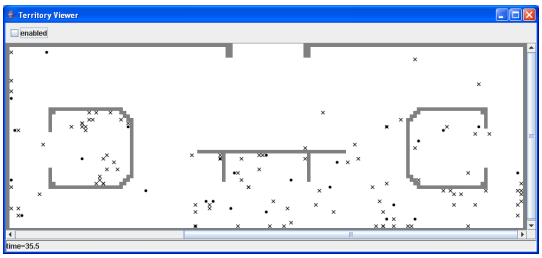


Fig. 3.13. A viewer screenshot during the execution

Parameter	**Value**
UAV visibility radius	15 s.u.
Target visibility radius	10 s.u.
UAV speed	10 s.u./t.u.
Target speed	5 s.u./t.u.
UAV initial fuel	500 f.u.
UAV initial ammunition	50
UAV fuel consumption	1 f.u/s.u.
Target initial energy	100 e.u.
α	0.1535
β	0.23
k	0.5
C_{dir}	1
T_{max}	100
U_{max}	100
MaxUAV	3
MaxTarget	3

Table 3.1. Model parameter values

UAV may be attracted by other targets despite the fact that it has already engaged with a target. In STR2 and STR3 an UAV focus its attention to a specific target once engaged.

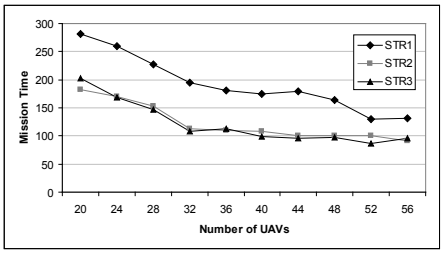


Fig. 3.14. Mission time vs. deployed UAVs

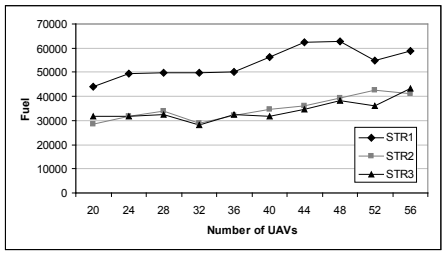


Fig. 3.15. Total amount of consumed fuel vs. deployed UAVs

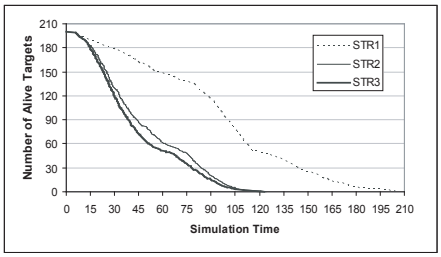


Fig. 3.16. Total number of alive targets vs. simulation time (32 UAVs)

From Fig. 3.14 and Fig. 3.15, it can be seen that, for the modeled scenario, using more than 32 UAVs does not result in a significant reduction in the completion time despite an increase in the total amount of employed fuel. Fig. 3.16 illustrates the evolution of the number of alive targets vs. the simulation time in the scenario where 32 UAVs are deployed.

3.3.5 Model scaling and simulation performance

A second set of simulation experiments addressed explicitly model scaling and corresponding simulation performance. The simulation model was run under strategy STR3 (see section 3.3.3). A larger grid territory of 450×60 cells with obstacles as in Fig. 3.12 and varying total agent population were split in an equilibrated way into three regions allocated to three theatres/federates, each running on a separate physical processor. Simulation parameters were those shown in Table 3.1. Agent population was scaled by keeping a ratio of 1 UAV per 6 targets, starting from an initial configuration of 270 UAVs and 1620 targets, and varying the scale factor from 1 to 10 times the initial configuration. For simplicity, all the upper side of the grid was used as entry area for the UAVs.

The wallclock time (WCT) required for completing the mission (i.e., all targets are consumed) was measured respectively in the distributed context (using the same physical architecture as described in section 3.3.4) and on a

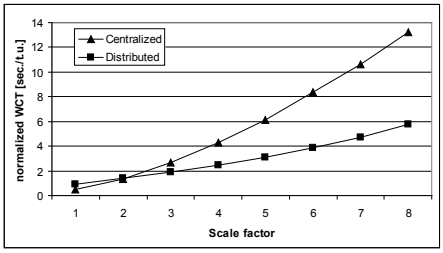


Fig. 3.17. Normalized WCT vs. scale factor

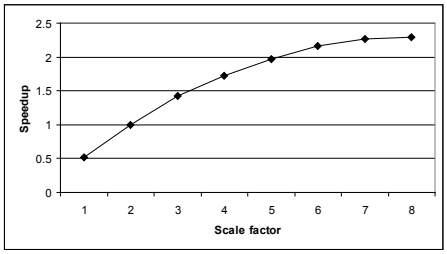


Fig. 3.18. Simulation speedup vs. scale factor (three processors)

centralized (serial) context (in this case the simulation model was run on a single machine of the distributed system, without HLA).

Fig. 3.17 shows the wallclock time vs. the scale factor. Each point is the mean value of five runs. To cope with small variations in the mission time during model scaling, in Fig. 3.17 the wallclock time is shown normalized with respect to the simulation mission time, i.e. normalized WCT=WCT/simulation mission time. For the scale factor 10, i.e. with agent population of 2700 UAVs and 16200 targets, the WCT of centralized and distributed execution of the simulation model was found to be about respectively 1551 sec and 619 sec. The mission time ranges from a maximum value of about 94 t.u. to a minimum value of about 83 t.u.. Lower values of the simulation mission time with respect to that shown in Fig. 3.16 are due to increased density of agent population and wider entry area which diminish UAVs average search time for targets.

Fig. 6.21 portrays the simulation speedup, i.e. the ratio between normalized values of centralized WCT and distributed WCT, vs. the scale factor.

As one can see from Fig. 3.17, both in the centralized and distributed case the wallclock time increases almost linearly vs. the scale factor, thus witnessing a good scaling of the simulation model.

Scalability and simulation performance can be related to the intrinsic concurrency degree of the simulation model. Despite an increase in the agent population, mostly of the exchanged messages are local. Only when an UAV/-target enters the boundary section of two adiacent regions, a network message

is sent by the local EnvActor to its neighbor peer (see Fig. 3.11) for updating the environment of the remote federate. Network communications also occur when a moving entity passes from a region to another. Message locality mirrors that for a given time advance granted by HLA, many local and concurrent messages exist to be processed without HLA intervention.

3.3.6 Related work

The approach based on Theatre over HLA can be related to known tools and methodologies for distributed simulation of multi-agent systems. Further related work can be found in references [JA06] and [LLT07].

Agent Framework Services (AFS) [JA06] is directly founded on the Actor Model [Agh86] which inspired the Theatre agent infrastructure. AFS characterizes by its dynamic agent distribution services which are based on two mechanisms: a *communication localizing mechanism* which co-locates agents that communicate intensively to one another, and a *load sharing mechanism* which moves agent groups from heavily agent platforms to lightly loaded agent platforms. An agent can migrate from its platform to another according to its own decision but also is moved by its platform according to communication locality. Agent platforms, though, negotiate actual agent migrations on the basis of their runtime load. Another fundamental service is ATSpace which supports agent discovery by properties. An ATSpace agent uses a *tuple space* where a tuple contains the name of an agent and its property, and supports agent discovery by tuple matching and tuple template matching. For complex search conditions, a search object can be sent to an ATSpace with the algorithm to be used for the selection process. ATSpace is a key for open agent-based systems and can be exploited for developing the (possibly decomposed) environment through which agents indirectly interact. AFS was experimented using an UAV simulation model [JA06] with conservative synchronization, witnessing reduced agent communication overhead and scaling. However, no speedup information is furnished with respect to a sequential version of the model. This is perhaps due to difficulties in managing thousands threads on a single machine. Interoperability was not a design issue and rests on the use of Java as the implementation language.

PDES-MAS project [LT01] defines a general framework for the management of the environment and shared state variables of agents. The framework is based on the concept of *spheres of influence* which characterize the immediate effect of agent events on related environment state variables. An algorithm is proposed which by tracking changes in (an approximation of) the spheres of influence of events is dynamically capable of decomposing an environment in sub-environment components, moving state variables to selected environment components etc. so as to reduce costs of accessing interested state variables. The approach is suggested to operate under an optimistic control structure. PDES-MAS concepts are felt interesting to experiment with in future Theatre

agent-based systems, e.g. using conservative synchronization which is more natural for highly reconfigurable systems.

Cougaar [KFC04] is a complex Java-based architecture integrated with HLA. It uses the concept of Society as a collection of agents. Agents in a society can be organized in (not distinct) Communities. Agent functionalities are provided by *plug-ins*. Plug-ins can share objects through a publish/subscribe API and the Blackboard collective memory store of agents. Blackboard objects can be used for communications. The Logic Providers are agent components which watch Blackboard activities and are responsible for messaging and Blackboard modifications. Messaging is out of plug-ins, which allows plug-ins to focus only on business logic instead of infrastructure details. Binders are used as wrappers to control communications among agent components and requested services. Being based on HLA, Cougaar favours interoperability with existing simulators. It has been used for developing such complex applications as UltraLog (military logistics), addressing robustness, survivability, scalability and security concerns. Policy is the primary means by which a system adapts and responds to changing conditions and attacks [**?**]. Agent mobility can be constrained (latency in the move) by the distribution of necessary policies required by the agent on the destination node. Execution performance rests to be documented.

HLA_AGENT [LLT07] is a recent tool developed for casting the SIM_AGENT legacy agent simulation toolkit [Sim] into HLA, with the major goal of backward compatibility with existing SIM_AGENT simulations. By providing additional information about which simulation entities are to be simulated by each federate, a developer can distribute a legacy SIM_AGENT simulation so as to make best use of available computational resources. The realization depends on RTI/C++ and it is assisted by a library which defines wrappers from and to RTI. Agent migration is simulated by object/proxy creation/deletion operations. A particular problem concerns concurrent access/update to shared attributes e.g. of the environment. Conflicts resolution, as in [**?**], can be achieved by divesting attribute ownership to RTI. In the Theatre UAV model, the environment is accessed only by messages (which are atomically processed) and consistency is automatically ensured. When an UAV agent plans to change position, it sends a timestamped message to the local EnvActor which replies to the agent after the move time is elapsed then, if needed, the agent is migrated to a different theatre. HLA_AGENT was experimented with a distributed version of SIM_TILEWORLD and its execution performance compared to the original sequential version of SIM_AGENT. A system of 50×50 cells, with a dynamic number of tile, hole and obstacle objects, and maximum number of 64 agents (reactive, i.e. network bound, in one test; deliberative, i.e. planning or CPU bound, with explicit plan time in another test) was simulated using a maximum of 16 cluster processors. Modest speedup was measured both for reactive agents and (better) for deliberative agents. The experience confirmed scalability of the simulation model but with an important overhead introduced by HLA.

HLA_REPAST [MT08] is a complex development similar to HLA_AGENT. The goal is to allow modelling of a multi-agent system using the popular sequential toolkit RePast [rep] and then offering the possibility of either executing the model on a single machine or, when it is too complex and computationally demanding, on a distributed context using a standard middleware. The reported simulation performances, using the same tileworld model, are similar to that obtained using HLA_AGENT.

Of course, results achieved using Theatre with UAVs cannot be directly compared with those of HLA_AGENT or HLA_REPAST with tileworld but the actor implementation over HLA seems to be more lightweight, lean and efficient, thus capable of delivering better execution performance to applications. Further work is required, though, for investigating the usage of Theatre in the modelling and simulation of more large and challenging multi-agent systems.

4

Supporting M&S formalisms through Theatre

4.1 Introduction

The Theatre agent-based architecture can be flexibly used as runtime infrastructure for various M&S formalisms. In this chapter, a specialization of actors as statecharts and the support of DEVS are discussed.

4.2 Hierarchical actors[1]

This section discusses the use of Theatre for the distributed simulation of discrete event systems (DESs) whose entities have a complex behaviours. Complexity is dealt with by exploiting statechart-based actors which constitute the basic building blocks of a model. Even if state-based formalism have been successfully used for specifying agent behaviours, drawbacks arise when each single agent may express a very complex behaviour. This means that such languages have to face the well-known phenomenon of state-explosion which is typically addressed by resorting to hierarchical and modular constructs.

Statecharts (see 2.2.2) are an extension of classical state transition diagrams which have such type of features. The basic mechanism consists in the possibility of nesting a sub automaton within a (macro) state thus encouraging step-wise refinement of complex behaviour. Statecharts have been successfully applied to the design of reactive event-driven real-time systems ([HP98],Selic:98,furfaro:06), as well as to modelling and performance analysis, e.g. ([VCA02, VCAA06]).

The type of statecharts used for modeling actor behavior are "distilled", in the sense that they permit only the or-decomposition. All of this complies with the basic assumptions of the adopted actor computational model where concurrency exists at the actor level but not within actors (see 3.2).

[1] based on [CFGN09b]

Distributed simulation is accomplished by partitioning the system model among a set of logical processes (theatres). Timing management and inter-theatre communications rest on High Level Architecture (HLA) services. In the following it is show a practical application of the approach through a manufacturing system model.

4.2.1 A modelling example

This section illustrates an application of the approach by considering a model of a manufacturing system. The example is adapted from ([VCA02, VCAA06]) where it has been handled by statecharts with and-decomposition and event broadcasting [HP98], and analytically studied by preliminarily transforming the model into a continuous time Markov chain (CTMC). In this paper the model is simulated. Obviously, simulation opens to the possibility of using probability distribution functions for event occurrences beyond the exponential one which is normally a prerequisite for building a CTMC. In addition, simulation can be exploited for investigation of more general properties about system behaviour. In the considered manufacturing systems (Fig. 4.1), two machines, respectively referred as Machine A and Machine B, operate in series processing the submitted jobs for producing a single product. Jobs are first processed by Machine A and then by Machine B. A bounded capacity Inventory is used for decoupling the operation of the machines thus reducing their wait times. A Robot is actually in charge of loading/unloading the two machines with jobs, possibly using the Inventory for temporary job buffering. Both machines and the robot may be subject to failures. Repairing from a failure is responsibility of an Operator. Machines, Robot, Operator and the Inventory are modelled as actors. Event broadcasting, which was present in the original model, is replaced by direct communications. In particular, because the robot bases its decisions on the operation state of the two machines and of the inventory, it gets directly notified by these components about relevant state changes. Analogously, when the robot, or one of the machines, goes in a failure state it directly asks the operator for being fixed.

Statechart models

Figures 4.2 4.3 4.4 4.5 depict the statecharts modelling respectively the behaviours of Machine, Operator, Inventory and Robot actors. The top states of all these statecharts have a default state named New, which is a leaf state, where each actor waits for the arrival of an Init message carrying initialization information. After being initialized, a machine (see Fig. 4.2) goes into state W where it waits for a job to be processed.

From W, it moves into state P when it has been loaded by the robot with a job. While residing in state P the machine processes the loaded job. At processing end, the machine moves into WU waiting for being unloaded. Both states W and WU have an entry action that consists in sending a Notify

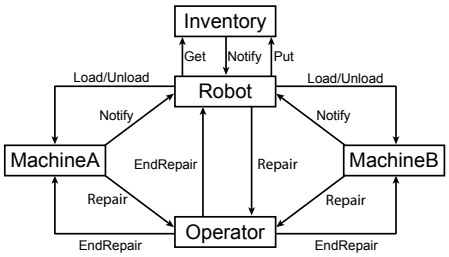


Fig. 4.1. A manufacturing system model

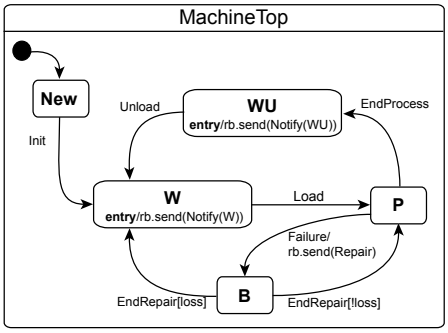


Fig. 4.2. Machine behaviour

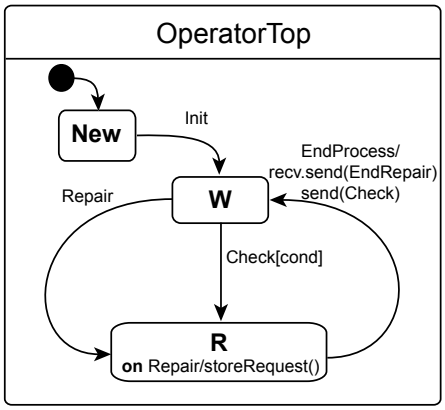


Fig. 4.3. Operator behaviour

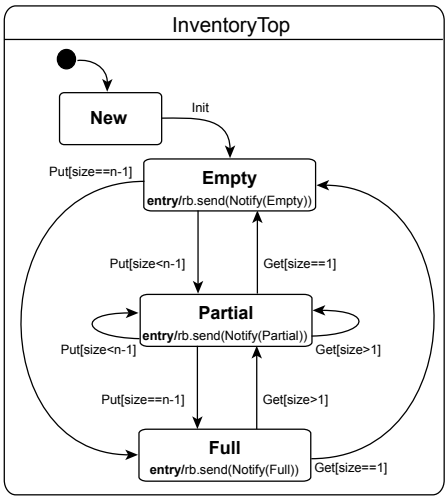


Fig. 4.4. Inventory behaviour

message to the robot for letting it know that the sending machine needs to be loaded or unloaded.

The message EndProcess is an internal message which a machine sends to itself for simulating the processing time (dwell time in P). During its stay in P, a machine can be subject to a failure in which case it moves to state B. As for processing end, failure is also modelled by another internal message which the machine sends to itself according to the next time to failure defined by a corresponding probability distribution function. When a Failure message is received, a request for being fixed is sent to the operator through a Repair message. After being repaired (arrival of the external message EndRepair coming from the Operator), the machine can return into state P for continuing processing of the interrupted job, or it can go back to W in the case the partial processed job is lost. The two possibilities are controlled by the boolean variable loss whose value is determined accordingly to the specified loss probability.

The behaviour of the Operator is depicted in Fig. 4.3. After being initialized, it waits in state W for a repairing request. As soon as such a request is received it moves into state R. The Operator resides in R for a dwell time that models the repairing time whose mean duration changes depending on the actor (a machine or the robot) that has made the request. While in R, other repair requests may arrive. They are handled by an internal transition whose action consists in storing them into suitable internal variables, thus postponing their processing. The completion of the repairing process is achieved with an internal message EndProcess to which the actor reacts by sending an EndRepair message to the relevant actor and a Check message to itself. Upon receiving a Check message, the operator inspects its internal

variables for checking whether there are pending repair requests. When faced with the decision of which entity to repair first, the operator chooses according to the following priority order: first machine Mb, then machine Ma and lastly the Robot.

The behaviour of the Inventory is portrayed in Fig. 4.4. It is a bounded buffer of capacity n, which can be 0 or a positive value. After initialization, the Inventory can be in one of the states among Empty, Partial or Full. Depending on the current available space, a Get/Put message can switch the inventory between Empty, Partial or Full as shown in Fig. 4.4. All of these three states have an entry action which consists in notifying the robot about the current inventory state.

The robot is the most complex entity as can be seen from the statechart of Fig. 4.5 which models its behaviour. While in state W, the robot waits for an operation to be exercised on the machines. Whenever the robot receives a Notify message it always updates its internal variables according to the received information. This is mirrored by the internal transitions of states W, P and B. In particular, if the robot gets notified when it is in state W, it proactively sends to itself a Check message. On the basis of the information about the state of the other components, if it is able to do some operation when the Check message is received, it switches to macro state P and sends to itself a Start message.

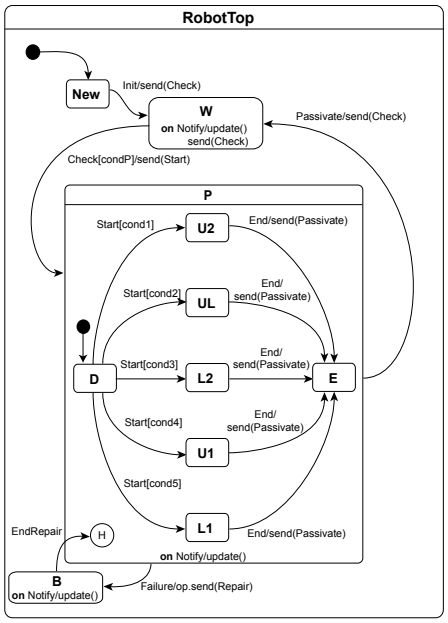


Fig. 4.5. Robot behaviour

```
cond1 = Mb is in WU;
cond2 = Ma is in WU && Mb is in W;
cond3 = Mb is W && (Inv is empty);
cond4 = Ma is in WU && (Inv is Full);
cond5 = Ma is in W;
condP = cond1 || cond2 || cond3 || cond4 || cond5;
```

Fig. 4.6. Environmental conditions for the robot

	Machine A	Machine B	Robot
Production rate	β_A=8,10 or 12	β_B=10	
Failure rate	λ_A=1	λ_B=0.5	λ_R=1
Repairing rate	μ_A=10	μ_B=15	μ_R=10
Loss probability	p_A=0.5	p_B=0.3	
Loading rate machine A			γ_{L1}=100
Unloading rate machine A			δ_{U1}=100
Loading rate machine B			γ_{L2}=100
Unloading rate machine B			δ_{U2}=100
Moving rate from m. A to m. B			α_{UL}=70

Fig. 4.7. Simulation parameters

This condition is reflected by the value of the boolean variable condP which is the logical or of various conditions as summarized in Fig. 4.6. State D is the default sub state of P which is left as soon as the Start message is delivered. At least one of the guards of the transitions leaving D is satisfied because each of them implies condP (see Fig. 4.6). The robot gives priority to unloading machine Mb (cond1 and sub state U2), then to simultaneously unloading machine Ma and loading machine Mb (cond2 and sub state UL), then to loading machine Mb (cond3 and sub state L2), then to unloading machine Ma (cond4 and sub state U1) and, finally, to loading machine Ma (cond5 and sub state L1).

The time spent by the robot in any operating state depends on the particular operation (see Fig. 4.7) that it is accomplishing. The internal message End is self-sent for witnessing the end of the operation, in which case the robot moves first into the E state and then sends itself a Passivate message whose arrival takes the robot from state P to state W where the behaviour repeats again.

The transition having Failure as the trigger message is an example of a group transition. It means that whatever is the internal sub state of P, the arrival of the Failure message causes the state P to be exited and state B to be entered, where the Robot requires to be repaired by the Operator.

While the Robot is in an operating state, it can fail (internal message Failure received). In this case the on-going operation is interrupted and the

operator is asked for intervention. Which event arrives first between End and Failure depends on the next time respectively for completing the operation and for failing.

The transition triggered by EndRepair causes the Robot to return into macro state P with history (see the shallow connector history H). This way, the actor returns exactly into the internal sub state of P which was current when P was last left off at the time of Failure.

For the purpose of experimentation, all the timed events in the system are assumed to be exponentially distributed. Fig. 4.7 summarizes the rates (number of events per time unit) used for simulation (as in [VCA02, VCAA06]). Loading/unloading rate of machines are shown as dwell times in the corresponding operation state of the robot.

Simulation results

The manufacturing model was simulated using the parameter values in Fig. 4.7, with the aim of validating the distributed runtime infrastructure of statechart-based actors. Simulation uses dense time. Each experiment lasts after a time limit of tEnd=5 105. The model was split in two Logical Processes (LPs)/federates assigned to two distinct processors (Pentium IV 3.4GHz, 1GB RAM) interconnected by a 1Gb Ethernet switch in the presence of HLA (pRTI 1516). One LP was assigned the machines and the Operator, the other LP the Robot and the Inventory. Into each LP is also present a Monitor actor for collecting useful information about the simulation. Each monitor has methods for capturing such data about system productivity, utilization of machines, robot and operator, losses in machines, average inventory size etc. In addition, every monitor sends to itself, at the beginning of the simulation, a timed message to be received at tEnd. Following the arrival of such a message, the actor displays collected statistical information.

The system model was studied in three cases: when machine A has respectively a lower/equal/greater production rate than B (see Fig 4.7). System properties were analyzed vs. the inventory bounded capacity which was varied from 0 to 20. Experimental results comply with those reported in ([VCA02, VCAA06]), but furnish more detailed information about system behaviour.

Fig. 4.8 portrays measured system productivity (number of unloads from machine B per time unit) vs. the inventory capacity. As one can see, starting from 0, an increase in the inventory capacity increases the system productivity until the system reaches full-busy condition. In this condition the system exhibits maximum parallelism among components, with the inventory which smooths out instantaneous differences in the production speed of the two machines. The smoothing effect is obviously greater when the production rate of machine A grows.

The positive effect of using a not zero inventory size can be checked in Fig. 4.9 which shows the waiting time for unloading machine A vs. the inventory

capacity. This statistic was achieved by summing up the dwell time of machine A in state WU, waiting for the robot to unload the finished product, and then dividing the sum for the simulation time limit.

For completeness, Fig. 4.10 illustrates the wait time for unloading machine B, which has priority with respect to unloading machine A. As expected, machine B has a small wait time.

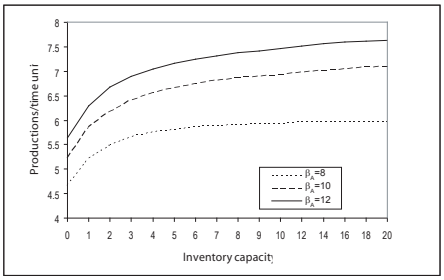


Fig. 4.8. Observed system productivity vs. inventory capacity.

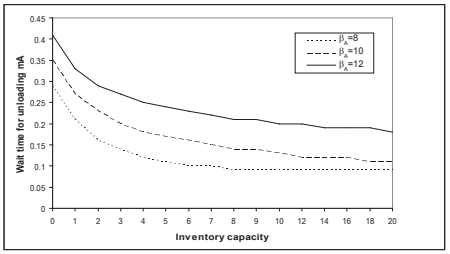


Fig. 4.9. Average wait time for unloading machine A

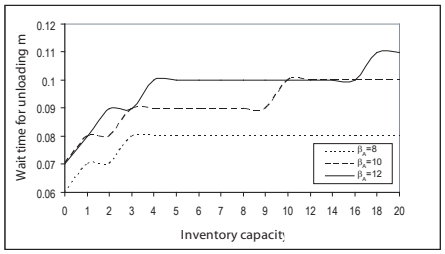


Fig. 4.10. Average wait time for unloading machine B

4.3 Actors for DEVS M&S[2]

In this section it is proposed a minimal and efficient Java framework - ActorDEVS- based on actors (agents) which enables modelling and simulation activities of discrete-event systems formalized by DEVS (see 2.2.1). A mapping of DEVS on to actors is described which leverages the expressive power of Parallel DEVS and makes it possible to build both atomic and complex hierarchical coupled models.

In particular, ActorDEVS is used under the perspective of the DEVS-World project, whose goal is the development of a net-centric modelling and simulation (NCMS) infrastructure having the net as the computer, thus favouring different levels of interoperability among research groups operating world wide. The section discusses some architectural scenarios for wrapping ActorDEVS in the DEVS-World infrastructure, opening to interoperability with other DEVS or (possibly) non-DEVS systems. The proposal clearly separates model and simulation concerns. An entire model is partitioned among a number of simulation nodes with *web services*, in a case, which act as the transport layer for inter-node message exchanges. A global coordinator with a minimal interface of operations governs the "in-the-large" simulation aspects.

4.3.1 ActorDEVS

ActorDEVS is a customization of the Theatre actor model (see 3.2) for supporting modelling and simulation of DEVS systems in Java.

The UML class diagram in Fig. 4.11 shows basic classes and their relations, useful for modelling (i.e., programming) and discrete event simulation of a single coupled model. The minimal framework rests on a few GoF design patterns. For example, the pattern strategy is employed to transparently weave the control structure, i.e. the simulation engine, to an application; the template method is adopted for structuring the `AtomicDEVS` abstract actor class which is the base for achieving, through inheritance, the concrete atomic components required by the application. Typed input/output ports of components are directly mapped on to messages. Toward this, the `Output<V>` and `Input<V>` derived classes of `Message` are introduced which are generic in the type `V` of the carried data. In particular, `Input<V>` is derived from `Output<V>`. Class `Output<V>` exploits the command and prototype design patterns. As a command, it has a send message to transmit its content to its destination actor, whose identity is established at configuration time. As a prototype, an output message is cloned in order to create an initialized copy of itself which is actually sent to its destination. Services `get()/set()` permit respectively to achieve/modify the data component of an `Output<V>` message. Method `linkTo(receiver)` allows an output message to be bound to a given receiver actor. The `Output<V>` class is provided of a recycler so as to avoid cloning

[2] based on [CFGN08]

when a consumed (i.e., already processed) output message is available for recycling. A `Timer` class is provided which inherits from `Message` and provides the notion of a timed message. A timer can be `set()/reset()` and the elapsed time since its setting or the remaining time before its expiring can be checked. When a timer is set, the identity of a timeout message, its receiver actor, and the relative expiration time must be given. Before expiring, a timer can be reset. A discrete time model is assumed. However, a dense time model could in alternative be used.

`DEVS_Simulation` embodies a discrete-event simulation control structure which manages two message data structures: an ordered queue of set timers, and a collection of instantaneous messages which have to be dispatched at current simulation time. Instantaneous messages take precedence with respect to timers. The control structure repeats a basic loop. At each iteration, a bag of instantaneous messages, if there are any, directed to a receiver atomic actor is formed and passed to an invocation of the `handler()` method of `AtomicDEVS`. If there are no pending instantaneous messages, the most imminent timer is allowed to fire thus advancing the simulation time. The corresponding timeout message is then passed to the invocation of the receiver `handler()` method.

The programming style is dictated by the abstract class `AtomicDEVS` which exports all the basic DEVS functions as abstract methods which a user-defined component class must override. The actual signature of functions is clarified in Fig. 4.12. As one can see, all transition functions return an int which codifies the next phase of the atomic component (from the point of view of the actor model, a DEVS component is a finite state machine built over the control states or phases of the component). The `delta_con()` method is concrete in `AtomicDEVS` and implements the default behaviour of the confluent function. Of course, a concrete component can redefine `delta_con()` to achieve a different behaviour.

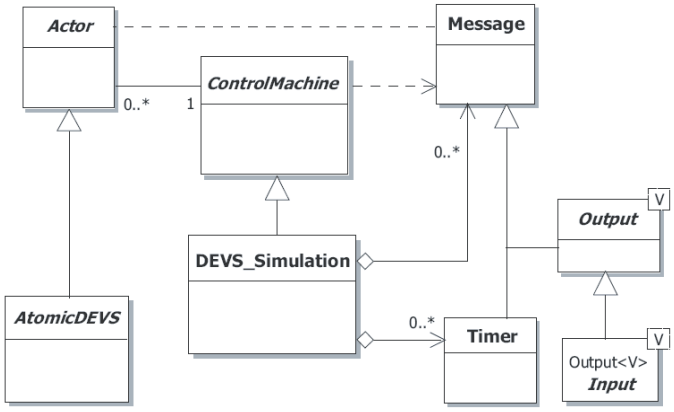


Fig. 4.11. UML Class Diagram of ActorDEVS basic Framework

```
public abstract int delta_int( int phase );
public abstract int delta_ext( int phase, long e, Iterable<Message> x );
public int delta_con( int phase, long e, Iterable<Message> x ){
    return delta_ext( delta_int( phase ), 0, x ); //default behaviour
}//delta_con
public abstract long ta( int phase );
public abstract void lambda( int phase );
```

Fig. 4.12. Signatures of DEVS Functions

A bag of inputs is an object created by the `DEVS_Simulation` engine, of an `Iterable<Message>` class. This way the modeller can navigate over the received messages (i.e. external events) by iterating over the x object.

An atomic component is initialized through the constructor whose parameter values are assigned to fields in the concrete atomic class. The `initialPhase(phase)` method must be called in the constructor for establishing the initial phase of the component. The `boolean acceptable( Iterable<Message> x )` method, always to redefine, returns true if messages in the bag x effectively belong to the input messages of the component; false otherwise.

The `void handler( Iterable<Message> x )` method of `AtomicDEVS` implements the DEVS semantics, i.e. it is in charge of making internal/external transitions and of checking simultaneity of an external event and an internal event (internal transition) in which case the confluent function is invoked. `AtomicDEVS` uses a timer and a timeout built-in message to enforce the temporal behaviour of the component. The class also exports the constant INFINITY for programming passive states.

4.3.2 DEVS-WORLD Vision

DEVS-World [Wor07] aims at developing a world-wide standard platform for modelling and simulation (M&S), promoting collaborative research and experimentation in the engineering, i.e. design, evaluation, implementation, deployment and execution of complex, scalable, dynamic structure systems [HW07] belonging to diverse and significant problem domains like biology and bioinformatics, environment systems, traffic simulation etc.

Novel in DEVS-World is the definition of a development methodology for supporting world-scale distributed *open* systems of systems M&S [Wor07]. Openness is a fundamental property which expands along different directions with different levels of integration and interoperability. A first level of integration is relevant to model interoperability. Many different implementations of DEVS simulators currently exist, and usually each of them uses a built-in modelling language often tied to a specific programming language like Java or C++. To cope with this problem, specific conversion tools capable of translating a DEVS model from a language to another can be realized. A more general

solution would be that of adopting emerging DEVS standard language such as DEVSML [Wor07]. Another direction of integration concerns interoperability at architectural level. In [Wor07] but also in [SPR+08] the proposed world-wide architecture is aimed at harmonizing heterogeneous models based on special-case DEVS tools, programming languages and engines, through the use of Web Services and SOAP dependent messages and other DEVS concepts (ports, simulators, coordinator etc.). Web Services are viewed as a world-wide *glue* enabling interoperation through DEVS/SOA mechanisms, with WSDL used for web services interface specification.

Besides standardization of models and simulation infrastructure, the definition of a standard simulation protocol is mandatory. The protocol (see 4.13) describes how a DEVS model should be simulated and how service/simulation engines should coordinate each other. Such a protocol opens also to a scenario in which both DEVS and non-DEVS simulators may (possibly) participate in a simulation.

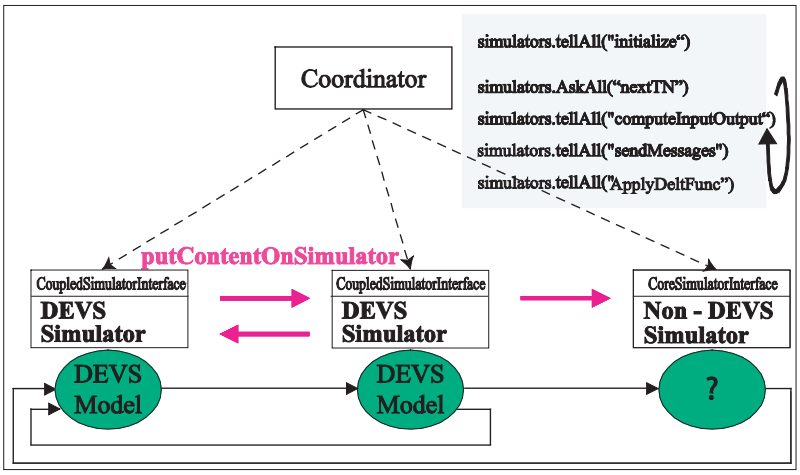


Fig. 4.13. Simulation protocol in a federation of DEVS and non-DEVS simulators

CoreSimulatorInterface (see 4.13) is the common interface to simulators. The term "core" means "essential" in that as long as a simulator implements this interface, it can participate in a simulation driven by a DEVS coordinator. In the case of DEVS-simulators, the *CoupledSimulatorInterface* is considered. This interface extends the core interface by providing other functionalities e.g. for adding/removing couplings among DEVS models.

CoordinatorInterface must be implemented by the coordinator. The coordinator is in charge of synchronizing the activities of the various simulators guiding them through the simulation control cycle. Basic phases of the simulation cycle are shown in Fig. 4.14.

Step	Description
nextTN	the coordinator requests that each simulator sends its time of next event and takes the minimum of the returned values to obtain the global time of next event
computeInputOutput	each simulator applies its *computeInputOutput* method to produce/gather an output that consists of a collection of *Contents* (i.e. port/value pairs)
sendMessages	each simulator partitions its output into messages intended for recipient simulators and sends these messages to these recipient simulators. Sending a message implies to call the recipient's *putContentOnSimulator* for any target simulator
applyDeltFunc	each simulator executes its *ApplyDeltFunc* method which computes the combined effect of the received messages and internal scheduling on its state. A side effect is in producing the time horizon gives back at the nextTN

Fig. 4.14. Simulation cycle phases

In handling simulation of hierarchical coupled models, a coordinator orchestrates a set of controlled simulators within it and, at the same time, can participate with peers in a coupled model above it. To allow such downward/upward facing interfaces, the *CoupledCoordinatorInterface* is introduced which extends both the *CoordinatorInterface* and the *CoupledSimulatorInterface*.

4.3.3 Wrapping ActorDEVS in DEVS-WORLD

This section highlights a service-based approach extending the Theatre/ActorDEVS architecture in order to meet requirements of DEVS-World. Pro-

vided extensions support architectural interoperability among heterogeneous DEVS simulators. The approach adopts previously described DEVS simulation protocol. At the moment, interoperability at modelling language level is not addressed. Each DEVS model is assumed to be implemented as a Java class complying with the ActorDEVS API [DAL08]. A *Coordinator* is introduced in order to coordinate the evolution of the overall simulation and it is in charge of implementing the DEVS simulation cycle (see Table 4.14). A *Configurator* makes it possible to configure the whole simulation system and start execution. An UML class diagram of system components is reported in Fig. 4.15.

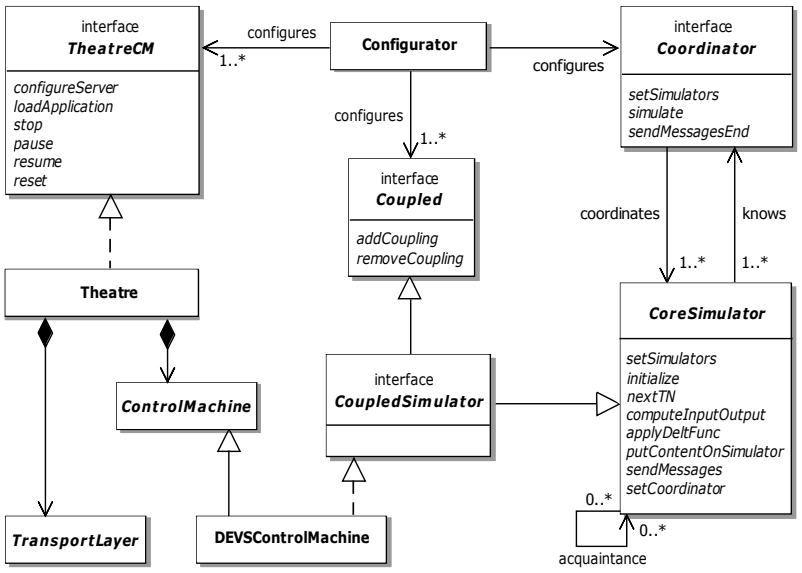


Fig. 4.15. Class diagram of system components

```
public interface Coordinator{
   void setSimulators(SimulatorInfo[] si)throws Exception;
   void simulate(long simulationTime)throws Exception;
   void sendMessagesEnd(Check check)throws Exception;
}
```

Fig. 4.16. Coordinator interface

The *Theatre* component and the *Configurator* are not exclusive of DEVS simulations, they are common to all actor-based applications. The *Coordinator* (see Fig. 4.16), instead, is tightly related to DEVS-World perspective. A *DEVSControlMachine* has been purposely developed in order to work in pair with the coordinator and be compliant with the DEVS simulation protocol. This control machine implements a *CoupledSimulatorInterface*-like (see Fig. 4.13) and behaves as a DEVS simulator. With respect to the approach proposed in [Wor07] the *Coordinator* is only concerned with the execution of the DEVS simulation cycle. In particular it does not manage coupling information among DEVS models allocated to different simulators. Such information is directly handled at simulator level. As a consequence, "local" and "external" couplings are handled in the same way. In addition, being in a net-centric context, the *Coordinator* must wait until all outgoing messages, i.e. inter-simulator messages, are received by recipient simulators before proceeding to the applyDeltFunc phase (see Table 4.14). This is ensured by Check messages (see Fig. 4.16) sent by simulators to the coordinator. Toward this, the `setCoordinator` method was added to *CoreSimulator* (see Fig. 4.15). Check messages are actually generated at the end of sendMessages phase and after external messages are received. *CoupledSimulator* interface (Fig. 4.15), which does not introduce further methods, extends both *CoreSimulator* and *Coupled interfaces*. This is to guarantee a clear separation of concerns among configuration (i.e. coupling management addressed by the *Coupled interface*) and simulation aspects (simulation protocol management addressed by the *CoreSimulator interface*).

In order to support the NCMS vision, a whole Theatre/ActorDEVS system, which can span from a single atomic model to a complex coupled model, is made usable through Web Services. Each system component is made available as a Web Service by means of specific objects called Wrappers. Client-side interactions are instead mediated by means of specific Proxy objects. It is worthy of note that in a service oriented architecture the roles of client and provider are not strictly defined, being possible for a same node to act as client or provider on the basis of the required/offered functionalities.

Wrappers and Proxies are transparently used. As a consequence, would e.g. Java RMI be used in place of Web-Services based protocols, only Wrappers and Proxies would be accordingly changed. Fig. 4.17 shows the architecture of a resultant Theatre/ActorDEVS system.

A *Code Server* is shared among theatres and it is used as a remote Java-class repository from which download the actor-based application to execute, i.e. in this case the DEVS models to simulate. Configuring and starting a simulation consists of four steps. The first step is devoted to setting-up the Theatre nodes by specifying the control machine, the transport layer to use and the code server IP address.

This is accomplished by exploiting the Configuration and Management Web Service (see the C&M-WS in Fig. 4.17). After the control machine is instantiated its functionality is made available as a Web Service which is au-

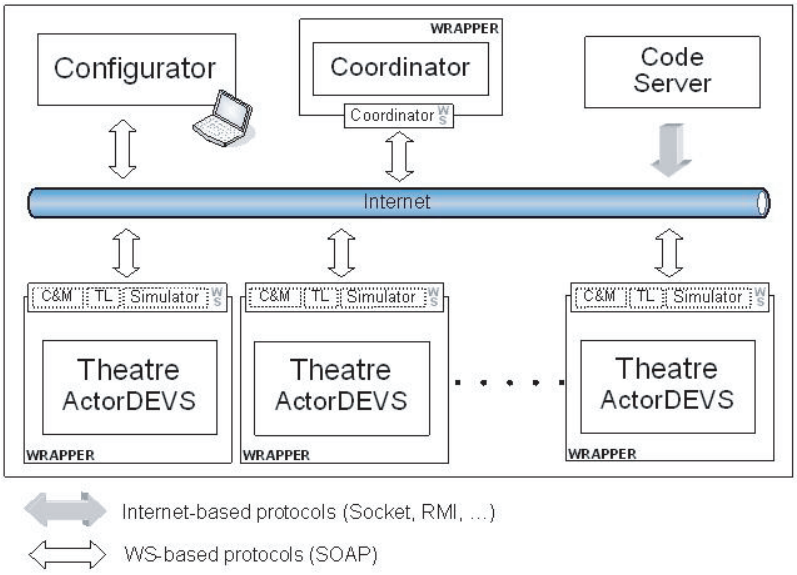


Fig. 4.17. Architecture of a Theatre/ActorDEVS system

tomatically published (see the Simulator-WS in Fig. 4.17). The *DEVSControlMachine* oversees message exchange with other simulators. As a consequence, the transport layer (see the TL-WS in Fig. 4.17) in this scenario is used only to manage inter-theatre control messages.

The second step consists in assigning to each Theatre the DEVS model(s) to simulate. A single model may correspond to an atomic or to a coupled DEVS component. The Java class name of each model requires to be specified along with the parameters possibly required by its constructor. This step is carried out by exploiting the C&M-WS and completes when models get assigned to target theatres, i.e. downloaded from the code server and instantiated.

The third step consists in establishing the necessary bindings among coordinator and simulator services (i.e. acquaintance relationships). In particular, a *CoordinatorInfo* object is provided to each simulator and a list of all *SimulatorInfo* objects, relevant to simulators involved in the federation, is furnished to each simulator and to the coordinator. An info object contains the name of the service and the relevant service endpoint address which is necessary to contact and use it. As stated above, each simulator has to know the coordinator in order to communicate information about the state of the current sendMessages phase (see Table 4.14).

The fourth step consists in defining couplings among deployed models in order to build the entire simulation model. This is achieved by invoking the method *addCoupling* onto simulators. Coupling information mainly contains

a couple of names, identifying the two ports to be connected. The first name is relevant to an output port of a DEVS component local to the simulator. The second name is relevant to an input port of a DEVS component which can be either local to the simulator or residing on a remote simulator. In the latter case, the name of the remote simulator is provided along with coupling information. A naming policy is required to distinguish ports belonging to different instances of the same model. In particular, full name of a port is assumed to be specified in the form *modelInstanceName.portName*.

At runtime, remote couplings get actualized by means of the so called *RelayPort* objects. Making a remote coupling implies linking an output port of a DEVS component to a relay port which, in turn, is logically connected to a remote input port. All of this makes the DEVS component unaware of network partitioning.

All data needed during configuration steps are contained in an XML file whose schema is reported in Fig. 4.18. In the current prototype system implementation, the `Settings` type is used only to contain the simulation time info. The `CodeServer` and `Coordinator` types contain information required to contact the relevant components on the web (e.g. service name, host, port). Other types are self-explanatory.

At configuration end, the Configurator may launch the simulation by calling the simulate method on the Coordinator which in turn triggers into execution the simulation control loop.

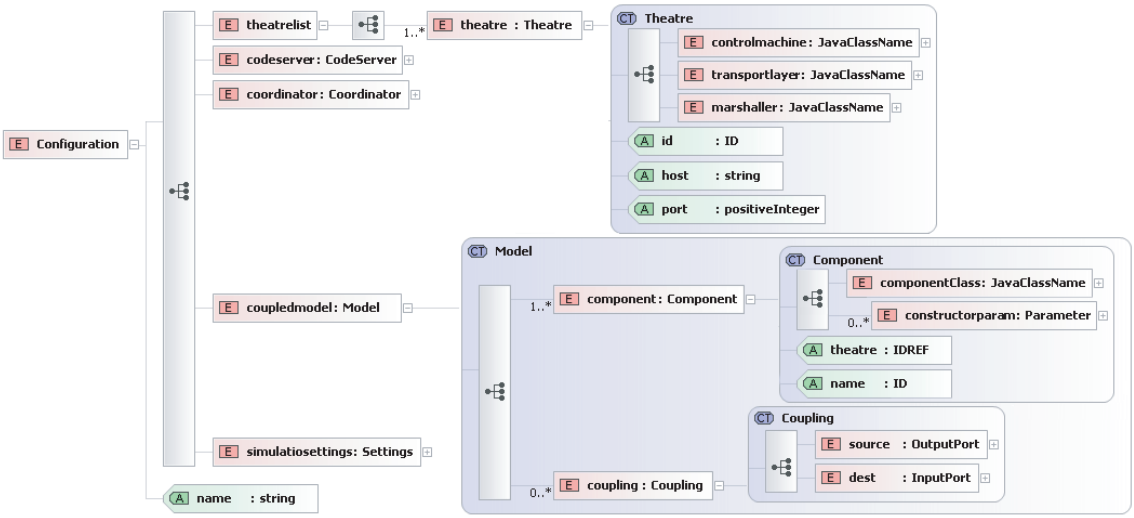


Fig. 4.18. XML schema of the configuration files

4.3.4 Variable structure system example

The achieved implementation of WS-based Theatre/ActorDEVS architecture was tested by modelling and simulation of a variable structure system based on server relocation [DAL08]. The modelled system consists of a collection (closed pipeline) of interconnected node components (see Fig. 4.19).

Each node receives from its environment a stream of jobs, stores them in a buffer (of unbounded size) and ultimately processes them using a number of server components. A system is assumed to work with a fixed number of servers. Servers cannot be dynamically generated because they model physical computing resources. However, a high loaded node can ask for a server to its neighbours. A dispatcher component in a node is in charge of handling the server relocation issues. Main difference between the model as handled in [DAL08] and here, consists in the achievement of structure dynamism.

In [DAL08], server components migrate from a node to another as mobile agents. In the scenario of this paper, though, servers do not migrate but port objects are created/destroyed dynamically in order to contact servers.

Asking for a server may return a server port through which a dispatcher can submit a job to a server allocated on a different node. As a consequence, server relocation is achieved by changing the number of servers a node can contact to process its jobs. Different strategies of server relocation can be considered (see later).

Fig. 4.19 depicts a three node system, together with input/output ports and connectors. Each node can direct useful statistical data to an external *Statistics* (*transducer*) component connected to the *StatOut* output port. When used, the *OverloadGenerator* can inject jobs randomly to any node.

Fig. 4.20 shows the internal structure of a node. Inter-node ports serve to send/receive an ask to/from a neighbour for a server (*ask-OUT?*, *askIN*), to send/receive a server to/from a neighbour (*moveIN?*, *moveOUT?*, *moveIN]*), or to send/receive back a no longer useful server (*sendBackOUT?*, *sendBackIN*). Fig. 4.19 shows *delegate* connections (represented by using dashed lines) within a coupled node. The shadowed *TimerToken* component in Fig. 4.20 is required only by some relocation protocols.

A high loaded node, that is a node with a pending job but without idle servers, asks for a server port to its neighbours. When the *Dispatcher* of a node receives a request for a server, it honours the request with a server port if at least one idle server is available. Otherwise the request is ignored. If no server ports are obtained, a node asks again for a server port after a certain time delay. Three particular strategies (Cicirelli et al. 2008) were considered about the way a node can handle external utilizable servers.

On-demand strategy - A node which achieves an external server, views it as an own server. Therefore, the protocol freely distributes server ports among nodes on a on-demand basis. It can be anticipated that this strategy makes it possible for nodes to behave in a selfish-way, possibly leading to an unbalanced distribution of server use.

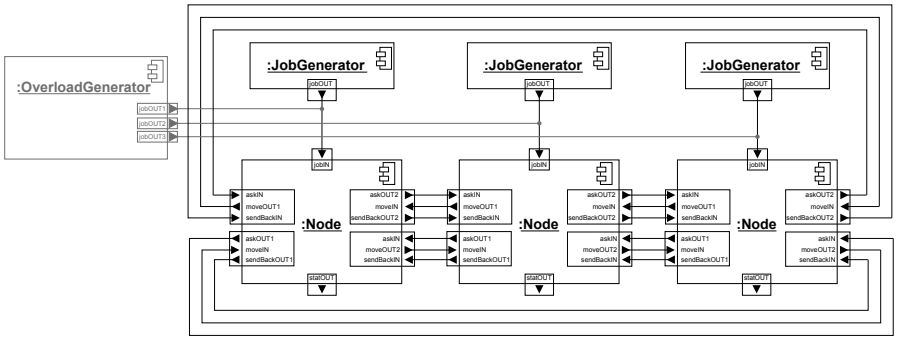


Fig. 4.19. A ring of three nodes

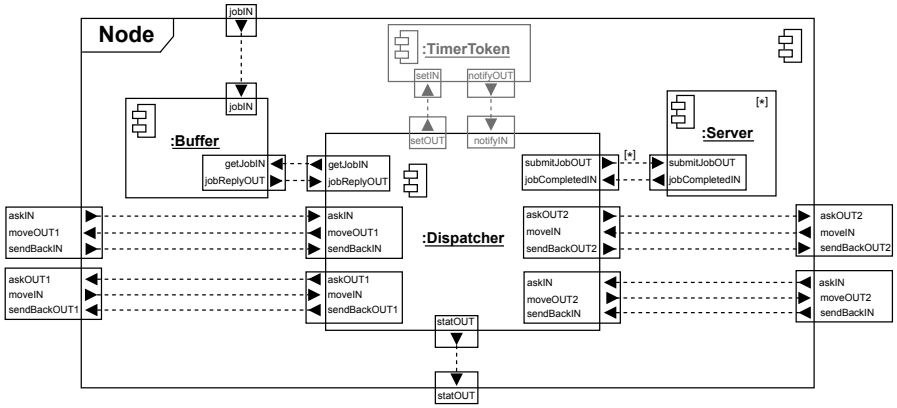


Fig. 4.20. Internal structure of a node

Debt strategy - A debt concept for server allocation is introduced. A node which receives a server port from a neighbour, annotates the identification of the furnishing node. As soon as the *Dispatcher* of a debtor node has no pending job but has at least one idle server, it tries to exhaust its debits by anticipating restitution of some server ports to its creditor nodes. Intuitively, the protocol attempts to avoid non uniform utilization of servers. *Token passing strategy* - One server port is used as a token which circulates upon the closed pipeline. A node receiving the token-server can use it if has a pending job but has no available local server. Otherwise, or after token usage, the token is forwarded to the next node in the ring. The strategy tries to anticipate a server request. A node which receives the token as well as server ports coming from neighbours, uses the token and sends back the other server ports.

4.3.5 Configuration, deployment and simulation

Some simulation experiments concerning the server relocation model described in the previous section were carried out by using two Theatre/ActorDEVS systems allocated on two Win platforms. Another Win platform was used to host the Coordinator, the Code Server and the Configurator. The experiments were directed to study the effects of overloads starting from an equilibrium situation. Simulation parameters which, under either On-Demand or Debt strategy, ensure the buffers size or equivalently the mean delay time of jobs is definitely constant and of a low value are as follows.

The job interarrival time is in the interval [2,4], the job size (which indicates the time needed to process the job) belongs to the interval [8,15]. The time delay a *Node* waits between two consecutive asks for a server was set to 1 time unit. The number of servers initially allocated to each node is 4. Starting from the equilibrium, the *OverloadGenerator* (see Fig. 4.19) is capable of injecting each generated job to a randomly chosen node. To respond to the overload, one additional server was introduced, whose management ultimately depends on the adopted strategy(ies). For instance, under On-demand or Debt strategies the extra server is initially assigned to a given node. In the Token passing strategy, instead, the extra server (its port) circulates in the pipeline ring. In this case, to avoid Zeno behaviours, the token which reaches the node where it was last used, is forced to wait one time unit before starting the next round. The job mean delay time (that is the time which elapses between the instant in time a job is received by Buffer and the subsequent time the job gets assigned to a server) was measured by the Statistics components. The investigated strategies for responding to overload were: Debt & Token, On-demand & Token, On-demand alone. The DEVS models relevant to *Node*, *JobGenerator*, *OverloadGenerator* and *Statistics* were deployed to the Code Server. A number of *Nodes*, varying from one to five, along with the relevant instances of *JobGenerators* were assigned to each Theatre. The *OverloadGenerator* and the *Statistics* were allocated on a single Theatre. The simulation time limit was set to $t_{END} = 10^5$. Different system configurations were actualized by specifying different configuration files. An excerpt of such a file is reported in Fig. 4.21. The configuration is relevant to a relocation system model made up of two *Nodes* allocated to two theatres. Only the Debt strategy is considered. Coupling information, common to all the configuration files, is used to build up the overall simulation model. In particular:

- each *JobGenerator* was coupled with the relevant *Node*
- each *Node* was coupled with its neighbors in the closed pipeline
- the *OverloadGenerator* was coupled with all the *Nodes*
- each *Node* was coupled with the *Statistics*.

Coupling information dictates system topology at configuration time. At runtime, on the basis of the adopted strategy, a *Node* may dynamically change the servers it actually contacts without resorting to the add/remove coupling

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration name="RelocationServers"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="./TheatreDEVS.xsd">
<theatrelist>
 <theatre id="PERSEUS8000" host="perseus" port="8000">
  <controlmachine name="theatre.DEVSControlMachine"/>
  <transportlayer name="theatre.transport.WSTransport"/>
  <marshaller name="theatre.marshaler.ByteArrayMrshlr"/>
 </theatre>
 <theatre id="HYDRA8000" host="hydra" port="8000">
  ...
 </theatre>
</theatrelist>
<codeserver url="http://orion:8989"/>
<coordinator name="Coordinator" host="orion" port="8080"/>
<coupledmodel>
 <component name="Node1" theatre="PERSEUS8000">
  <componentClass name="relocation.Node"/>
  <!-- number of servers -->
  <constructorparam type="java.lang.Long" value="4"/>
  <!-- token disabled -->
  <constructorparam type="java.lang.Boolean" value="false"/>
  <!-- debt enabled -->
  <constructorparam type="java.lang.Boolean" value="true" />
 </component>
 <component name="Node2" theatre="HYDRA8000">
  <componentClass name="relocation.Node"/>
  ...
 </component>
 <component name="OverloadGenerator" theatre="PERSEUS8000">
  <componentClass name="relocation.OverloadGenerator" />
 </component>
 ...
 <coupling>
  <source theatre="PERSEUS8000" port="Node1.sendBackOut2"/>
  <dest theatre="HYDRA8000" port="Node2.sendBackIn1"/>
 </coupling>
 <coupling>
  <source theatre="PERSEUS8000" port="Node1.askOut2"/>
  <dest theatre="HYDRA8000" port="Node2.askIn"/>
 </coupling>
 <coupling>
 ...
</coupledmodel>
<simulatiosettings>
   <simulationtime>100000</simulationtime>
</simulatiosettings>
</Configuration>
```

Fig. 4.21. An excerpt of a configuration file

mechanism. Simulation experiments (see Fig. 4.22) indicate that the combination of Debt & Token strategies minimizes the job mean delay time when compared to the other strategies.

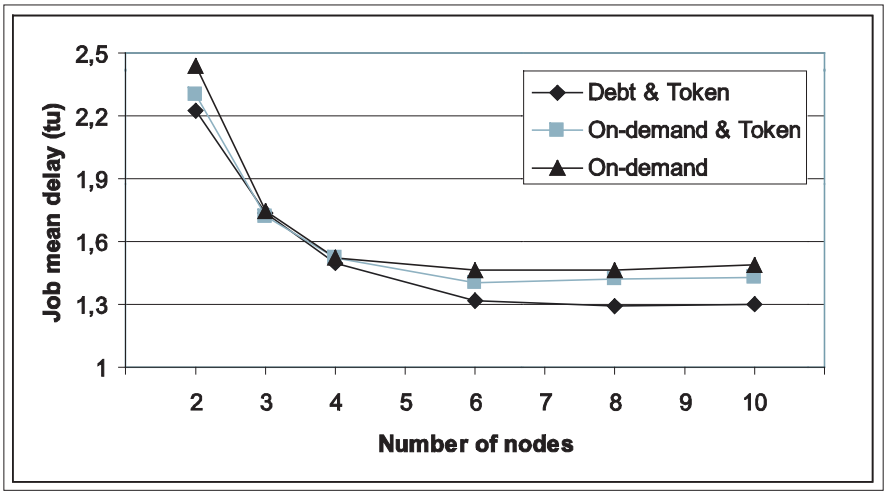


Fig. 4.22. Job mean delay time vs. number of nodes

5

Theatre over Terracotta[1]

5.1 Introduction

This chapter describes a work on high-performance agent-based modelling and parallel/distributed simulation using the Terracotta middleware (see section 2.3.2). The research goal is an exploitation of Multi-Core Clusters (MCCs) [BHJ+10] where each machine is made up of one or more multi-core CPUs. MCCs represent a cost-effective paradigm shift in modern microprocessor architectures, but they challenge the development of suitable software infrastructures capable of exploiting the available computing power. In this work the adopted approach is based on the *Theatre* agency (see section 3.2). Multiple theatres/JVMs are capable of executing in truly parallelism when allocated to distinct cores of an MCC. The approach is novel in that it combines three levels of concurrency: (a) cooperative concurrency among non pre-emptive light-weight actors local to a same theatre/JVM, (b) pre-emptive concurrency among actors executive thread and JVM i/o interface threads, and (c) truly parallelism among JVMs executing on distinct cores of an MCC. Terracotta is used so as to provide a transport layer to communicating actors belonging to different theatres, and to support the implementation of a conservative synchronization algorithm [Fuj00] which coordinates theatres' global time advancement. Terracotta transparently clusters the JVM by a network-attached heap memory holding shared object graphs. Trasparency depends on aspect oriented programming and bytecode injection at class loading time. This chapter reports a performance study concerning modelling and simulation of a scalable predator/prey model under different MCC distributed/parallel execution scenarios. The developed model naturally requires agent mobility and relies on a coordination strategy among competing predators which is based on the evolutionary minority game [CZ97].

[1] based on [CFGN10b, CFGN10c]

5.2 Design issues

Terracotta data mechanisms were carefully exploited for supporting message passing among theatres allocated for execution to different JVMs. The realization purposely reduces message traffic and lock overheads with Terracotta server which are very important to improve the execution performance. Not only it was immediately abandoned the idea of turning actors/agents as global objects, which would imply too much Terracotta intervention at each actor change, but as a fundamental design principle it was realized to keep hidden to Terracotta both actor and message contents. As a consequence, Terracotta use was limited to transport layer services and global time coordination. Actors remain local agents to a theatre/JVM, but can move from a theatre to another. Fig. 5.1 shows the architecture of a typical Theatre system based on Terracotta. Each theatre is hosted by a JVM instance and represents a Logical Process (LP) of the application. TTI stands fore Theatre to Terracotta Interface component, and provides services to theatres for sending/receiving remote messages.

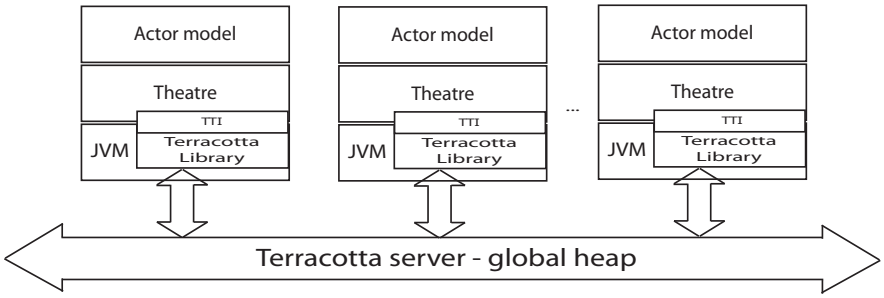


Fig. 5.1. Architecture of a Theatre system based on Terracotta

Intra-theatre communications are directly supported by the control machine services. Inter-theatre communications depend on FIFO channels (buffers) as depicted in Fig. 5.2 which refers to three interacting theatres. Channels are unidirectional and point-to-point, that is there exists a buffer for each partner theatre.

A view to the internal details of a theatre is portrayed by Fig. 5.3. The organization was designed according to a *lock striping* strategy [GPB+00], i.e. by introducing distinct locks for the various buffers and for guarding specifically the data used for time coordination.

TTI contains the local heap versions of clustered input buffers, output buffers and data supporting the algorithm for time coordination. Each input buffer is associated with a *reader thread*. The writer thread of output buffers coincides with the control machine thread. A reader thread is nor-

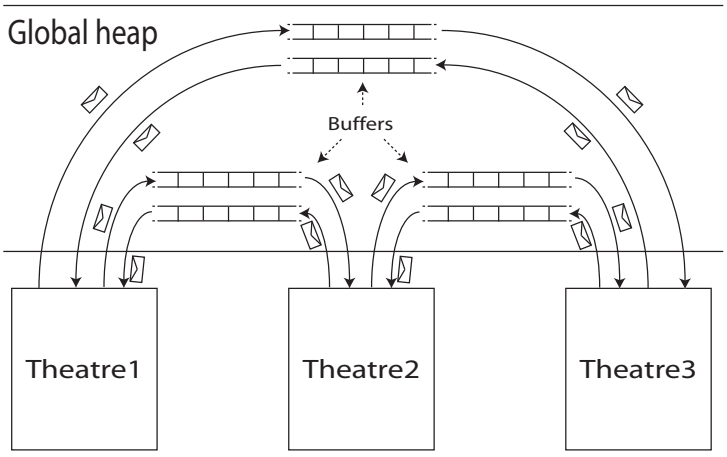


Fig. 5.2. Theatre transport layer based on Terracotta

mally blocked, waiting for incoming messages. The arrival of external messages awakes a reader which then gets a block of messages and schedules them (according to message timestamps) upon the message queue of the control machine. To reduce lock synchronization burden, sending/receiving operations of external messages involve *bag* of messages. In fact, working with buffers always implies taking a local then a global lock with Terracotta server intervention. At its awaking time, a reader thread empties its input buffer. Similarly, an output buffer is filled and at *due* times, i.e. when the output local bag (see Fig. 5.3) is full or a time advancement is requested (see later in this section), flushed upon the output buffer.

A critical design issue concerns the transmission of an inter-theatre message or migrating actor. To minimize automatic Terracotta *mirroring* task, objects (i.e. messages or actors) are sent as raw *byte arrays*. In particular, a customization of Java standard serialization mechanism is used. Customization refers to the fact that an actor field in a message/actor object is never serialized in the standard way. Rather, URL information about the actor are actually transmitted, that is the specification of the originating theatre where the actor resides. To avoid even accidental modification actions, a sent remote object is copied (i.e. marshalled or serialized) and then the copy is put in the output buffer. After that, reference to the copied object is locally lost. Similarly, at the time an object is received, it gets unmarshalled (deserialized) and then the reference to received byte array object lost. A deserialized actor which finds a proxy version of itself in the receiving theatre copies its status back on the proxy object through minimal recourse to reflection. Then the proxy becomes a normal actor.

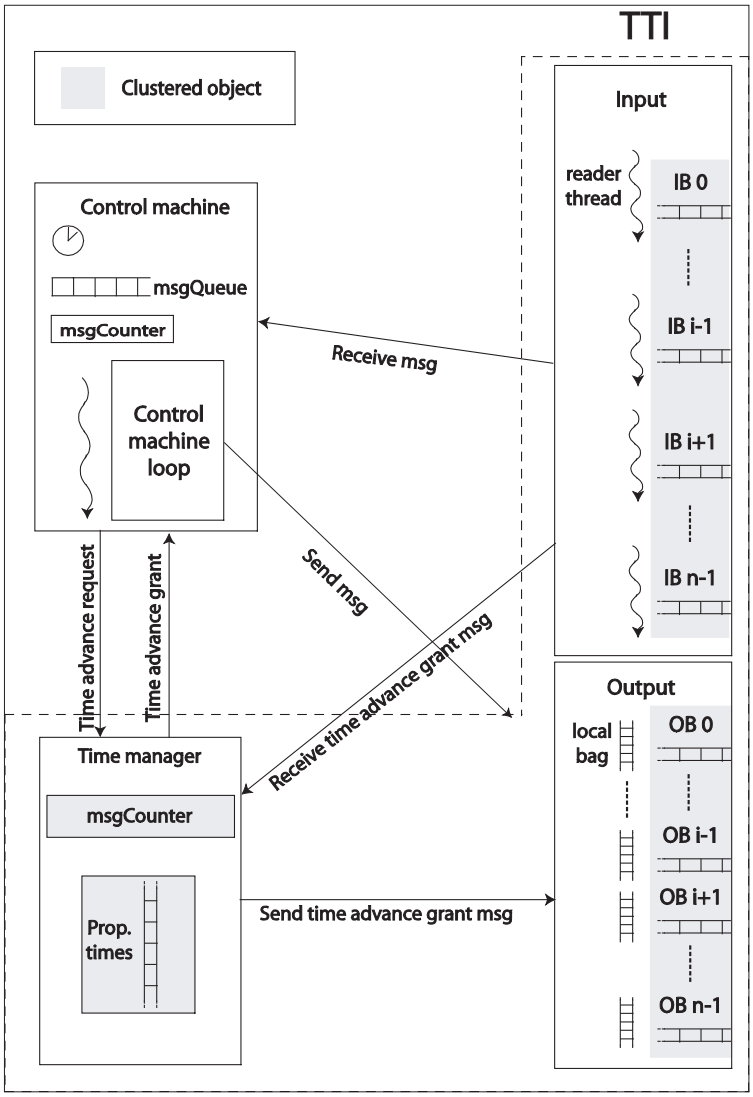


Fig. 5.3. Details of internal organization of a theatre

Reader threads serve also as a *callback* mechanism which implicitly Terracotta invokes. When a message arrives in an input buffer, it is the reader thread which has the responsibility of action prosecution into the receiving theatre. In the normal case, a received message has only to be scheduled at its occurrence time in the message queue of the control machine. A different course of actions happens when time management is involved.

A distributed conservative synchronization algorithm (zero *lookahead*) [Fuj00] was implemented where every theatre can play the role of time manager or time coordinator. Clustered data for time management include a *msgCounter* of sent/received messages and an *array of proposed times*, in which each theatre has its own entry. The local msgCounter of control machine (see Fig. 5.3) is incremented at each remote send, decremented at each remote receive. The use of global and local msg counter reduces the need to take the clustered lock at each change of the counter of messages.

When the control machine has no more messages to process at current simulation time, it asks for a time advancement to the (absolute) timestamp of the most imminent message in the message queue. Such a time is the theatre *proposed time*, and it is stored in the clustered proposed time array at theatre index. In addition, the value of the local msgCounter is added to the global msgCounter. In the case the requesting theatre finds global msgCounter is zero (meaning no in-transit message exists) and all theatres have proposed a time advancement, the theatre qualifies as current time manager. Then it finds the minimum of proposed times and broadcast such a new time to all the theatres who have requested advancement to the same minimum time. As a consequence a TAG (Time Advance Grant) message with new minimum global time is prepared and sent to relevant theatres.

As in [CFN08], for generality reasons, the theatre simulation time is actually a composed time made up of a triple ¡*virtual-time*, *generation*, *step*¿ where virtual time is the model time. Remaining fields are useful to establish precedence constraints among concurrent messages. Concurrent messages generated by a given (causal) message can be tagged with the same virtual time of the cause but with a new value of the generation (step is 0). If a message must occur at the end of current generation, the step field 1 can be used which acts as the least significant bit of the compound time notion. As shown in [CFGN10a] message generations can be exploited in a case to establish a partial-order for resolving conflicts among actors e.g. in boundary cells of adjacent regions of a partitioned agent space. An example of a message to be heard at the end of current generation is an actor migration message. Migration can be actually accomplished when no more contemporary messages exist in the originating theatre, directed to the migrating actor. By scheduling the migration message to occur at current time and generation but at step 1, it is guaranteed that the message will be processed at the end of current generation, i.e. after having processed any other contemporary message.

5.3 A Predator/Prey model

As a case study, a complex multi-agent system based on predator-prey model [JG00, PL05] was developed and simulated. Agents move over a territory and have a visibility radius. Predators have a greater visibility radius than preys. With respect to other predator-prey models, the goal of a predator here is to attack a prey in order to eat it thus consuming its vital energy.

Once a prey has been perceived, a predator has to decide about heading toward it. Information about the number, position and status of both surrounding preys and predators may influence the hunting behavior of a predator.

The territory is assumed to be a bi-dimensional grid where each cell is supposed to be large enough to host a certain number of predators and preys. Information about (part of) the hunting ground is handled by a specific environment actor (EnvActor). Predators and preys are modeled as agents (PredatorActor and PreyActor) whose behavior states how the respective entities dynamically react to information about the surrounding environment. This information is obtained by exchanging messages with the EnvActor.

Predators normally explore the territory using a random walk until some preys enter the visibility radius. The way in which a predator decides to move to a prey depends on the adopted *hunting strategy*. Two different strategies were considered and compared. The first one is a *greedy strategy* which drives a predator to move to a visible zone in the territory that contains the greatest number of preys. No coordination is actually used by predators. The second strategy introduces a coordination among predators which is based on a minority game. Once a prey is seen, the minority game is used to establish which predators cooperate to capture the prey and which one remain free to engage another hunt. Also preys make a random walk for the exploration of the territory avoiding directions leading to predators.

5.3.1 Greedy strategy (str1)

A PredatorActor follows the simple behavior expressed by statechart in Fig. 5.4 In the Explore state, the predator receives a message EnvInfo from the EnvActor containing information about other actors residing in the visible surrounding area. The information is used to determine the next move. In addition, a timed message is sent to the EnvActor in order to communicate predator next position which will cause the EnvActor to reply with a subsequent EnvInfo with updated territory information. When the predator arrives in the cell of a prey, the predator switches to Capture state from where it hits the prey and eventually follows any prey movement. On prey death the predator comes back to Explore status.

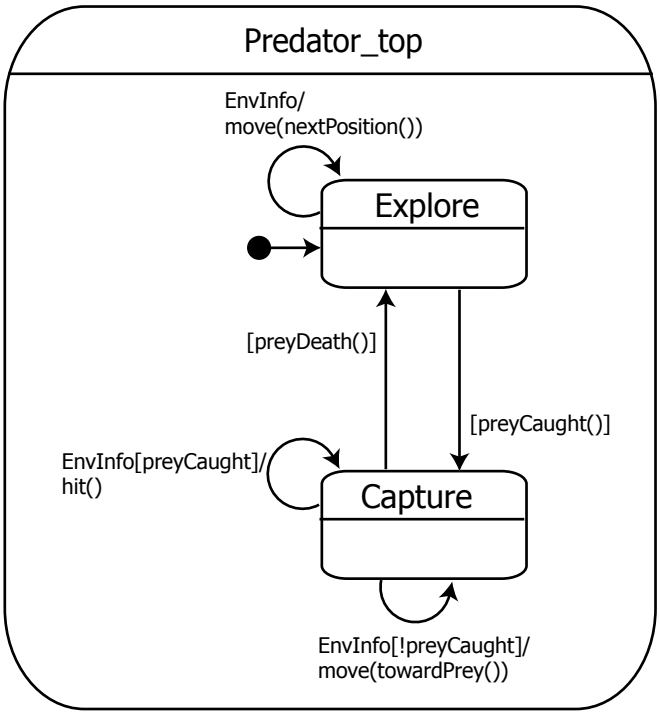


Fig. 5.4. Predator actor behavior in the greedy strategy

5.3.2 Minority game strategy (str2)

Minority Game (MG) is an evolutionary game originally proposed in [CZ97]. In its basic formulation, there are N (supposed odd) players that make a choice between two options at each turn. Winners are those that have made the choice which is in the minority side, i.e. the one chosen by at most $(N-1)/2$ players. Each player is initially fed with a set of strategies that it may use to calculate its next choice on the basis only of the past M outcomes of the game recorded in the player memory. Since there are only two possible outcomes, M is also the number of bits needed to store this information. The number of possible histories is of course 2^M. Players rank their own strategies based on their respective capability to predict the winner side. Every player associates each strategy with a virtual score, which is incremented every time the strategy, if applied, would have predicted the minority side. At each turn, a player uses the first ranked strategy. The players earn a reward at every step they choose a winner strategy. At the beginning of the game, the content of player memories is randomly initialized and the strategies are equally ranked.

A modification of MG, also used in this work, consists in blaming bad strategies (and also the players applying them) by decrementing their virtual score. In another scenario, the minority side is identified by using a cutoff value k different from $(N-1)/2$ [JHZT99]. Other variations generalize the MG by introducing an acquaintance relation among players. Each player owns a local view of the game, i.e. it will win or loose because of the decisions taken by players who are in the acquaintance relationship (Local Minority Game) [RC05].

In the hunting strategy proposed in this paper, a Local MG with a cutoff value k was adopted. Let $O = \{-1, +1\}$ be the set of possible outcomes of the game where -1 corresponds (see Fig. 5.5) to the Explore state (i.e. the predator does not contribute to capture a prey) and $+1$ to the Capture state (i.e. the predator goes to capture a prey), P be the set of predators involved in a local MG and I a subset of natural numbers corresponding to game steps. Let $h : P \times I \rightarrow O^M$ be the function modeling the history of predators, i.e. $h(p, i)$ returns the last M outcomes of the games of a predator p preceding a given game step i. Let $S = \{s_1, \ldots, s_n\}$ be the set of all allowed strategies ($n = 2^{2^M}$) and $s_j : O^M \rightarrow O$ be the strategy function which guesses the next winner side by looking at the game history. Let $V_{s_j} : P \times I \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers, be the virtual score given by a predator p to the strategy s_j at game step i. The initial value $V_{s_j}(p, 0)$ is set to zero for every predator. The play outcome of a predator p at game step i is $s_j(h(p, i))$ where s_j is the strategy having the highest virtual score, i.e. $\forall s_t \in S \setminus \{s_j\} : V_{s_j}(p, i) \geq V_{s_t}(p, i)$. The winner side at step $i \in I$ is computed as:

$$ws(i) = \begin{cases} +1 & \text{if } nc(i) < k \\ -1 & \text{otherwise} \end{cases} \tag{5.1}$$

where k is the adopted cutoff value and $nc : I \rightarrow \mathbb{N}$ counts the number of predators which in the step i gave $+1$ as play outcome. The reward $r : P \times S \times I \rightarrow \{-1, +1\}$ of a strategy s_h for predator p at game step i is calculated as: $r(p, s_h, i) = s_h(h(p, i)) * ws(i)$ and the relevant virtual score is updated as: $V_{s_h}(p, i + 1) = V_{s_h}(p, i) + r(p, s, i)$.

Fig. 5.5 shows the behavior of a PredatorActor according to local MG. While exploring the territory, a PredatorActor stays into the Explore status. As soon as it recognizes a prey which is not already engaged by other predators, it engages the prey and goes into the Manager status. Here the predator first checks for the presence of other predators within its visibility radius and then asks them to play a game by sending Invitation messages. The manager predator remains into the Invitation_Reply status until all predators reply with an Accept or Decline message. When all the replies are received, the game may begin: the manager sends a Play message to the game participants (i.e. predators which replied with Accept) and then wait for collecting the game results in the Wait_Result status. As soon as all the results arrived, the manager (i) plays its game, (ii) evaluates the winner side and the reward, (iii)

communicates winner side and reward to all players and then switches to the Game_End status. The game can be repeated a number of times in order to refine the virtual score of the strategies. All of this is mirrored by the presence of the playAgain() function as guard of outgoing arcs from the Game_End state. When the playAgain() becomes false, on the basis of the outcome of the last own game the manager comes back to the Explore state or goes to Capture where starts moving towards the engaged prey in order to eat it. In the Capture state the engaged prey becomes the only interesting entity for the predators. When the prey is eventually caught the predator hits it. The exploration begins again after the prey dies. The behavior of the predator in the Player state is dual to that of the manager. A predator becomes Player on accepting an Invitation message which contains both the identity of the game manager and that of the prey to catch. The default state of Player is Wait_PlayMsg where the player waits for the Play message which triggers a state transition to Wait_MG_Outcome. Before this, the player evaluates its own game outcome and then sends its result to the manager. When receiving the winner side and reward, the player goes to Game_End state. Here another game can be repeated or, on the basis of the own game result, the Explore state or Capture state is entered.

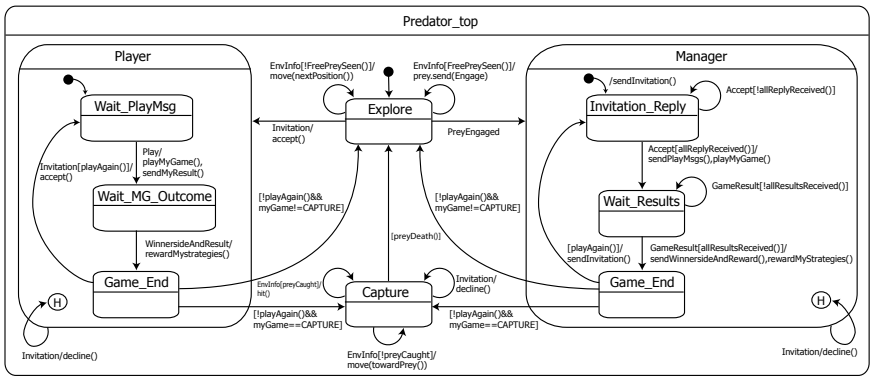


Fig. 5.5. Predator behavior in the MG strategy

5.3.3 EnvActor behavior

The role of an EnvActor is handling the environmental information that may be of interest to local agents. Besides the local portion of the mission area, the EnvActor maintains an updated snapshot of the environment parts that are not handled locally but which may fall in the visibility radius of a moving entity. Fig. 5.6 (similarly to Fig. 3.9) portrays boundary details between adjacent (sub) environments. The EnvActor exchanges messages with its neighbor

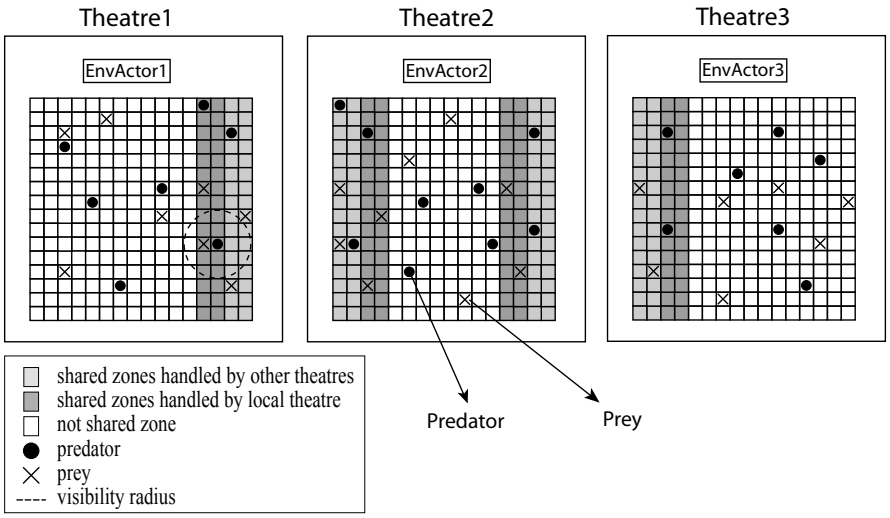


Fig. 5.6. An example of territory partitioning for distributed simulation

peers by sending them updates every time a local event causes a change in a shared zone handled locally, and by receiving updates from the other EnvActors notifying a change in a shared zone handled outside. When a PreyActor or a PredatorActor wants to move, it sends to the EnvActor a timed message with next position. The time-stamp of the message mirrors the time needed to reach the destination cell. If the position is under the influence of the EnvActor, the latter replies with a message containing information about the area in the visibility radius of the sender. Otherwise the EnvActor migrates the sender to an adjacent theatre and asks the relevant EnvActor to reply to the migrating actor. It should be noted that the described protocol makes PreyActors and PredatorActors unaware of distributed simulation concerns.

5.4 Simulation experiments

The multi-agent predator-prey model described in the previous section was simulated on three Win7 (64 bit) Intel i7 CPU 960, 4-core, 3.20 GHz, 6GB RAM, interconnected by a Gigabit Ethernet switch. The adopted Terracotta version was the 3.2.1ee on top of a Java HotSpot 64-bit Server VM version 14.3. This version of Terracotta limits the number of JVMs per federation to be at most 10. The considered hunting ground was of 3204x800 cells populated by an equal number of predators and preys, ranging from 20000 to 240000, randomly deployed. Each model execution lasts until all preys disappear from the territory. The simulation time of this event constitutes the model hunting time (MHT). The goal was estimating the simulation performance of the

predator/prey model and that of the actor infrastructure over Terracotta. Different execution scenarios, characterized by different model partitioning schemas and hardware/software configurations, were considered.

5.4.1 Strategies performance

As a metric for comparing the two hunting strategies of predators (see sections 5.3.1 and 5.3.2) the model hunting time (MHT) was chosen. Better strategy minimizes MHT. A preliminary set of experiments were accomplished for the MG strategy, by using different values of the number of game steps, from 5 to 30, always played at each detection of a new prey. It emerged that despite the augment in the number of game steps, the local MG strategy results in almost the same value of the MHT but, as expected, it requires a greater value of the WCT. As a consequence, the str2 strategy was evaluated by using the following parameters: cutoff k=10, M=2 (two bits of history), number of initial game steps equals to 5. This initial number of game steps serve as *training* for the predators, so as to refine the available score strategies. After initial training, each predator plays only one game for each new prey, in order to decide if hunting it. Fig. 5.7 collects the measured MHT vs. predator population for the two strategies. The MHT decreases as the number of predators and preys augments. This is because as the density of predators-preys increases it becomes easier for a predator to catch a prey. The strategy based on local Minority Game (str2) outperforms the other simple greedy based strategy (str1). For simplicity, remaining experiments on the performance study will be based on the minority games only.

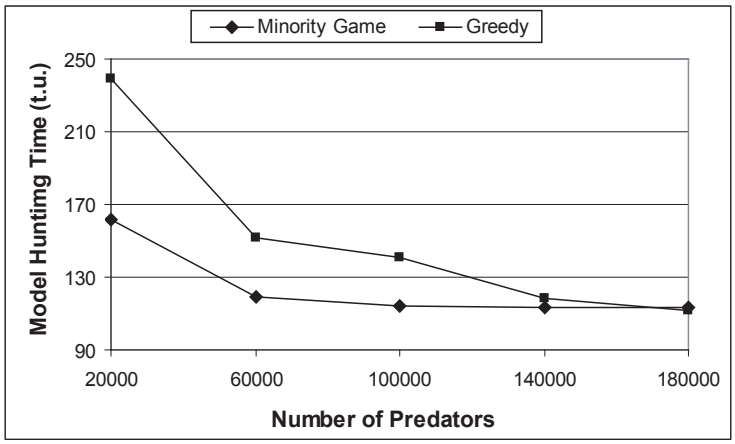


Fig. 5.7. Model hunting time vs. predator population

5.4.2 Simulation performance

Execution performance was evaluated by considering the WCT required for completing a hunt, i.e. all preys disappear from the territory. Obviously, WCT is influenced by the value of MHT. As a consequence, the value of WCT is normalized with respect to the model hunting time (*normalized* WCT = WCT/MHT).

Fig. 5.8 reports the normalized WCT vs. the number of predators in the case of a sequential simulation. This scenario refers to the case of one single theatre which is assigned the entire simulation model. Terracotta is not used and one single core was enabled on the computing node. The curve witnesses that the model scales well as the number of agents increases. These results were used for evaluating the relative speedup with respect to the other parallel/distributed scenarios.

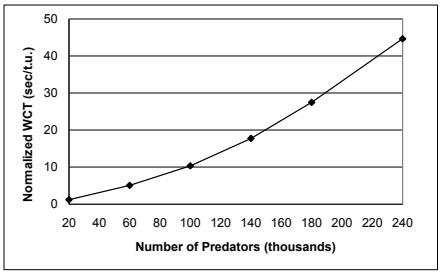


Fig. 5.8. Sequential simulation: normalized WCT vs. predator population

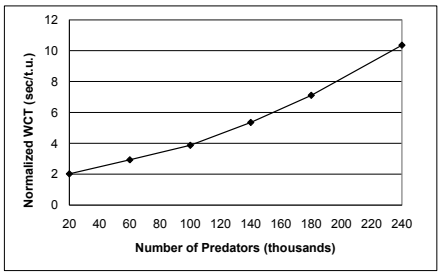


Fig. 5.9. Normalized WCT vs. predator population in a multi core cluster configuration

Figures 5.9 and 5.10 show normalized WCT and the achieved speedup vs. the number of predators in the case a multi-core cluster (MCC) configuration is considered. In particular, the scenario refers to three computing nodes

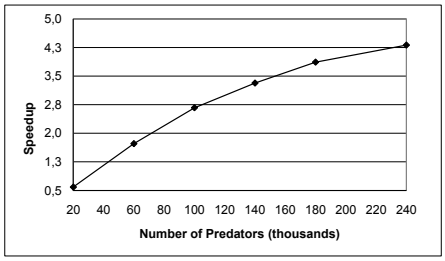


Fig. 5.10. Speedup vs. predator population in a multi core cluster configuration (3 machines, 9 applicative cores)

with four cores each. The model was equally split among nine theatres with a territory region of 356x800 cells assigned to each of them. A server array of three *active* instances of Terracotta was used. In this configuration, clustered data is automatically and equally split among the active servers in the cluster thus fostering high system availability. All of this avoids the bottleneck of managing shared heap data arising when one single active Terracotta server is used. Obviously, for performance issues, system configuration supporting shared-data persistence or fail-over mechanisms was not considered. In the scenario of 240000 predators, the above MCC configuration was able to reduce simulation time by nearly 1.5 hours (sequential simulation) to about 20 minutes.

Fig. 5.11 and 5.12 report respectively the normalized WCT and the speedup vs. the number of predators in the case a mere multi-core configuration is used. This parallel scenario consists of one single computing node with all the 4 cores enabled. One instance of Terracotta and three theatres were used. The simulation model was partitioned into three regions of 1068x800 cells and each region was assigned to one distinct theatre.

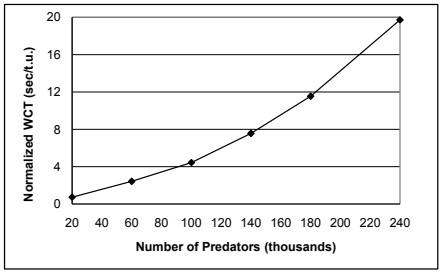


Fig. 5.11. Normalized WCT vs. predator population in a pure multi core configuration

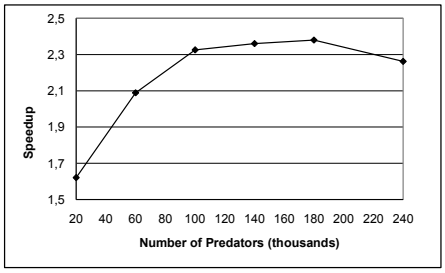


Fig. 5.12. Speedup vs. predator population in a pure multi core configuration

As another configuration ("pure" distributed), a scenario composed by three computing nodes each with one core only enabled, was considered. Model partitioning was the same as for the pure multi-core scenario, i.e. the model equally partitioned into three regions assigned to three distributed theatres. A server array composed by three *active* instances of Terracotta was used. Performance of this simulation scenario is reported in Fig. 5.13 and Fig. 5.14. These figures also show information about the performance achieved by enabling 2-cores in each computing node

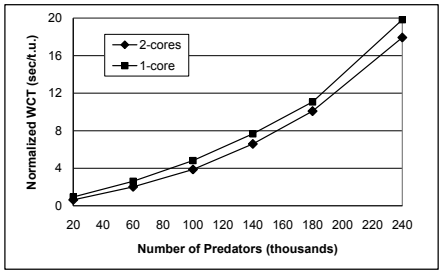


Fig. 5.13. Normalized WCT vs. predator population in distributed scenarios

As one can see from Fig. 5.12 and Fig. 5.14, the speedup diminishes in the case of 240000 predators. Moreover, differently from what one would expect, the speedup is lower in the "pure" multi-core scenario (Fig. 5.12) than in the "pure" distributed scenario (case 1-core in Fig. 5.14. In order to highlight the underlying behavior in the two configurations, simulations were profiled. Profiling was aimed to evaluating the overhead due to Terracotta with respect to the overall simulation. As described in section 5.2, Terracotta is used for inter-theatre data exchanges (*send* operation) and for managing time advancement (*time advance* operation). In particular, during a inter-theatre send, data exchange is achieved by putting data (*put* operation) within an output buffer (see section 5.2). A time-advance operation lasts until a grant message is received by the theatre that asked an advance of simulation time. In Fig. 5.15

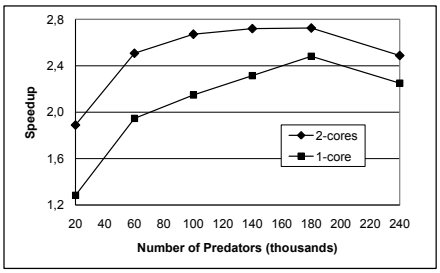


Fig. 5.14. Normalized WCT vs. predator population in distributed scenarios

and Fig. 5.16 is portrayed the real time needed to carry out the operations previously mentioned. The *Send-Put* curve refers to the time needed by send minus the overhead due to Terracotta. It is worth nothing that (i) despite the obtained speedup, in the multi-core scenario Terracotta performs better. In fact, both the send and time-advance operations require less time with respect to the mere distributed configuration, (ii) as expected, the send time is dominated by the time required to put data inside the shared heap, (iii) the send time suddenly increase, in both cases, when a higher simulation load is considered.

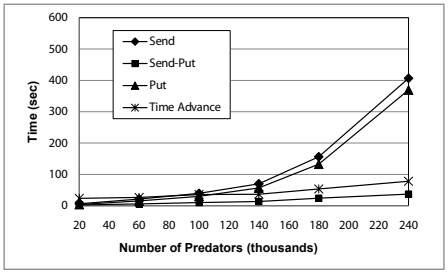


Fig. 5.15. Timing behavior of mere multi-core scenario

Fig. 5.17 portrays another view of the overhead times in the two simulated scenarios. Here relative computational times (i.e. the ratio between considered times in the parallel scenario and those of the distributed scenario) are shown. The curve *Sent MSGs* in Fig. 5.17 refers to the ratio between the numbers of exchanged messages that, as expected, tends to 1.

From Fig. 5.17 it emerges that, even if the number of sent messages remains almost the same in the two scenarios, the time spent in the *Send-Put* operations (without Terracotta intervention) increases in the multi-core configuration. Therefore, the multi-core configuration incurs a penalty during the simulation although Terracotta performs better (curves *Put* and *Time Advance* always remain below 1). All of this mirrors the fact that in a multi-

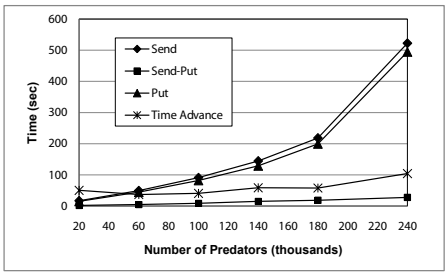


Fig. 5.16. Timing behavior of mere distributed scenario

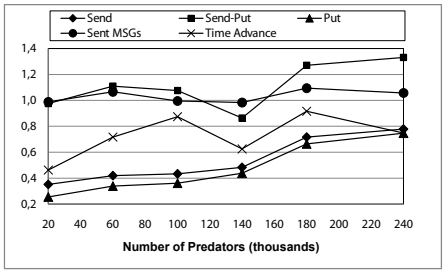


Fig. 5.17. Ratio between computation times in the parallel scenario and the distributed one

core scenario [BHJ+10] the overhead for managing concurrent accesses to the physical memory increases with the number of cores.

6

Distributing situated multi-agent systems[1]

6.1 Introduction

In the context of discrete-event simulation [ZPK00], the operation of a system is represented by a chronological sequence of events. Each event occurs at an instant in time and marks a change in the state of the system. The size and complexity of systems which are usually modelled as discrete-event systems (DESs) is ever increasing. Modelling and simulation of such systems is challenging in that it requires suitable specification languages and efficient simulation tools. Agent-based modelling [Woo02, Fer99, Jen01, OZ04] is a computational approach that models and simulates the interactions of autonomous and reactive entities with each other and their local environment to predict higher level emerging behavior. Distributed simulation [Fuj00] is ofter required to cope with the high resource demands of large system models with an inherent asynchronous behaviour like agent-based models. For distributed/parallel simulation a model is split into a network of interconnected concurrent Logical Processes (LPs), each maintaining and processing a disjoint portion of the simulation state.

The "spatial" nature of a broad range of studied phenomena and modelled systems (e.g. in biology, sociology, wildfires etc. domains) leads to the concept of *situated agent* [Fer99], i.e. an agent that owns spatial coordinates and is embedded in a spatial environment (territory). Situated multi-agent systems (MASs) are systems of agents whose behavior is strongly influenced by their position in the environment [BMS02]. Although there is an agreement about the importance of environments for MASs [WVP+05], in most cases the environment is not integrated in MASs or its responsibilities are minimized. As a consequence, a rich potential of applications and techniques that can be developed using MASs is not handled properly. Popular frameworks such as Jade, Jack, Retsina or Zeus reduce the environment to a message transport system or broker infrastructure. Well-known methodologies such as Message,

[1] based on [CGN10]

Prometheus or Tropos offer support for some basic elements of the environment; however they fail to consider the environment as a first-class entity [WVP+05].

The simulation of situated agents introduces particular challenges which are not addressed by standard parallel discrete-event simulation (PDES) models and techniques. Basic problems are tied to handling a large shared state, the agents' environment (territory), which is only loosely associated with any agent. This shared state introduces a bottleneck which limits the speedups that can be attained [Log07][PS09].

The following approaches can be traced in the literature on the subject. In [LT01] the distribution of shared state upon the LPs is based on the metaphoric concept of *spheres of influence*. Spheres of influence are dynamically determined on the basis of the mutual interactions among entities in the system. The goal is to distribute the shared state in order to favor nearness between an agent and the information it requires during execution. In [LLM+05] shared data is maintained in a tuple-space and an optimistic simulation algorithm is adopted for driving the simulation. The tuple-space is partitioned by following a hierarchical schema based on the spheres of influence so as to avoid bottleneck in managing the shared data. The same approach is exploited in [VMT09] where range query operations are allowed on shared data. In [vVJP09] is proposed an agent-based distributed architecture for the simulation of air-traffic systems. Handling of spatial data is achieved by partitioning the territory and allocating each single partition to a different LP. A partition, along with the agents situated on it, determines the work-load of an LP. No details are provided about the strategy used to ensure consistency and to avoid conflicts on shared data, e.g. occurring when different agents located on different LPs try to modify the same spatial data.

This chapter proposes an original approach to distributing situated agent-based models with the goal of simplifying modeling tasks while avoiding bottlenecks and fostering system scalability. The approach is based on the assumption that an agent privileges accesses to information located in its immediate "vicinity" (i.e. its neighborhood) without inhibiting the ability to manipulate or access information wherever located. To this end, the concepts of *visibility radius* and *action radius* are introduced. Such radiuses are designed to delimit the area within which an agent can efficiently read and change the status of the territory and the state of agents located in it. The territory is partitioned and distributed among various LPs. The portion of the territory (region) assigned to an LP and the set of agents located in it, constitutes the workload of the LP. The approach avoids the use of system-wide locks by purposely exploiting time management concerns to achieve data consistency issues. Such mechanisms are used to implements a framework which provides the notion of *stage* for the Theatre (see chapter 3 architecture.

In section 6.5 is described a concrete implementation of these mechanisms focused on distribution of RePast model using the Theatre architecture over HLA

6.2 Distributing spatial environments

In situated multi-agent systems, each agent is "embedded" into a spatial environment (territory) and has *partial control* and *partial observability* about the space "surrounding" it. An agent can perceive a partial view of the state of the spatial environment, and may act in order to change it. Even if the environment issues could be explicitly addressed by the modeller, there exists a general agreement on the fact that it is more convenient to manage environment issues as a "system-wide" concept thus promoting it as a "first class entity" [WVP+05], i.e. directly accessed by the multi-agent infrastructure.

Agents can interact with the spatial environment by following two main different approaches. The first is based on asynchronous communication, i.e. message passing. The second relies on the use of synchronous communication, i.e. method calls. The former requires a more complex coordination policy among the agents in order to ensure atomicity and consistency in read/write operations on the environment. Consider two competitive agents that want to read a property of a resource from the space and to change it on the basis of the read value. In the asynchronous scenario, each agent sends a *read* message to the space to know the state of the resource. The environment replies to both agents with the same value and the agents modify the value by sending a *write* message. The final state of the resource remains unclear because it is determined by the last processed *write* message. This could also result into an inconsistent state of the resource. A specific coordination among the two agents should be taken into account by the modeller. Another drawback of this approach is relevant to the higher number of messages which require to be handled during runtime and which can impair simulation performance.

The above considerations guide toward the adoption of a synchronous approach. However, by itself the synchronous approach does not fully resolve the problem, it is also required to guarantee that the read and write operation must be executed atomically.

In a sequential model it would be quite easy to guarantee atomicity e.g. by using lock mechanisms or by enforcing atomic agent actions to be interleaved. In a distributed scenario similar approaches could be also exploited but they could have an heavy impact on the performance. Moreover further mechanisms (e.g. remote method calls) are required.

In a distributed context, the territory is a *"huge shared variable"* of a concurrent system. Frequent remote accesses of agents to the territory, can be the basis for a *"bottleneck"* that degrades system performance and scalability. Furthermore, assigning the whole territory to a single LP and distributing agents among various LP, implies that the scalability of the system is constrained by the memory of the LP that contains the territory.

In the approach proposed in this chapter, spatial environment is arranged as a bi-dimensional grid, which is divided in spatial *regions* (see Fig. 6.1). Each region is allocated to a different LP. In addition, each situated agent is allocated to the LP that manages the region it belongs to.

The approach favors local synchronous access of the agents upon the territory. Naturally, splitting the space among different LPs requires agent *migration* when an agent moves from a region to another one.

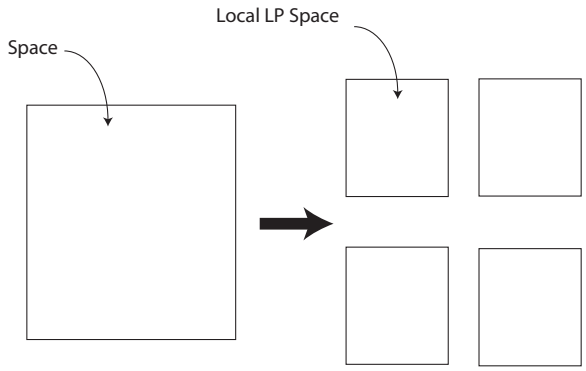


Fig. 6.1. Environment space partitioning

6.2.1 The problem of distributed shared state

The overall state of a situated multi-agent system is composed by the union of the state of each agent and the state of the territory. By considering the territory as a grid of cells, the status of the territory is the union of the states of the various cells.

An agent could be able to perceive and act upon the state of all the environment but more often it needs to read and write only its "*neighbourhood*". The neighbourhood of an agent located in a cell, is defined in terms of a **visibility radius** and a **action radius** centred on the cell. "Visibility" and "action" refer to read and write operations respectively.

In the proposed approach, operations 'inside the neghibourhood' and remote operations are explicitly distinguished. In the following, mechanisms are described that provide an efficient execution of operations in the neighbourhood realized by using a synchronous approach even in a distributed scenario. In section 6.3.4 the approach is extended for supporting remote operations in an asynchronous way.

Splitting the space ensures that if the agent's neighbourhood entirely falls in a single region, local synchronous access remains guaranteed. When agent visibility or action radius fall outside the local region, remote read/write operations and coordination among LPs are required. To avoid remote operations, a copy of the edge portion of a region is replicated in *adjacent* LP(s) which manage contiguous regions of space. Such a portion is referred to as a *border*

of the region. In figure 6.2 the gray part highlights border areas of a territory split between two LPs.

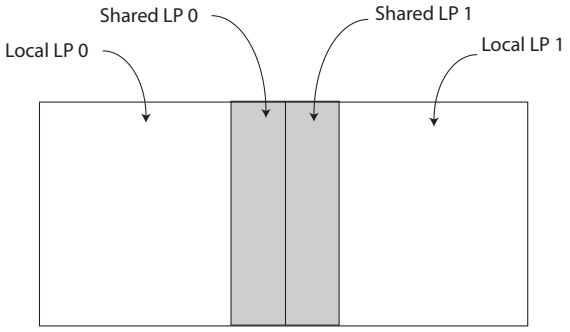


Fig. 6.2. Boundary regions of two adjacent LPs

The border area is made up of two distinct parts (see figure 6.3): a *local border* part, locally managed by the LP, and a *mirror border* part which is a replica of adjacent LP local part. Agents located in a border area, along with their states, are also mirrored. Each agent located in a local border part is reflected in the mirror border part of an adjacent LP through a *proxy* version.

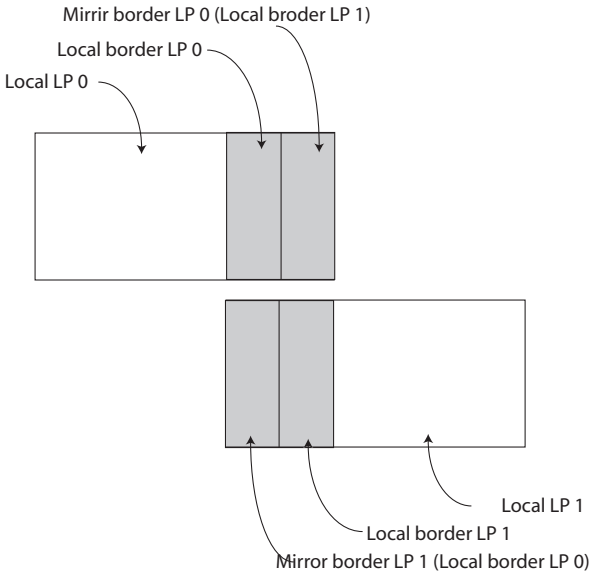


Fig. 6.3. Detailed view of border regions

The width of both local and mirror borders is set to the maximum between visibility and action radiuses. This ensures that within the neighbourhood read/write operations will be performed locally, i.e. in the same LP where the agent resides.

Boundary regions obviously creare problems of keeping consistency in border areas. An update to a boundary local cell must be also reflected in the mirror part of an adjacent LP. A second kind problem concerns conflict resolution during concurrent accesses to shared space cells by border agents executing in different LPs. In the approach proposed in this chapter, such problems are resolved by a mechanism based on a *composite logical time* notion as described in the following.

6.2.2 A mechanism for conflict resolution

As has been previously stated, the border area of a region needs to be carefully managed. Every state change in a local border must be reflected in the mirror side of a neighbour LP. The state update should be accomplished at suitable real time so as to guarantee data consistency during simulation. In addition, a border area should be guarded against conflicts as explained below.

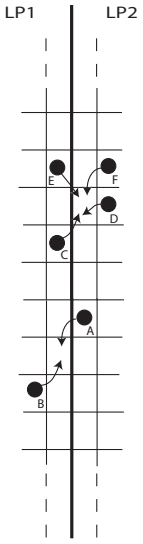


Fig. 6.4. A situation of potentially conflicting agents

For the sake of simplicity the following examples refer to a scenario in which each cell in the territory can contains at most one agent. In figure 6.4 a scenario in which a group of agents is involved in cell conflicts is shown. A *naive* solution to prevent conflicts should be that of statically establishing

an execution order among all the agents deployed over border area. Such a solution suffers of two drawbacks: ordering the agents results in a unfair system evolution in which some agent is always preferred over others. The second problem concerns a poor exploitation of the inherent parallelism of the multi-agent system. Indeed, as one can see in Fig. 6.4, not all agents are conflicting each other. More than one agent can be allowed to move at the same time. For example, agents `A` and `B` in Fig. 6.4 are completely independent from the others.

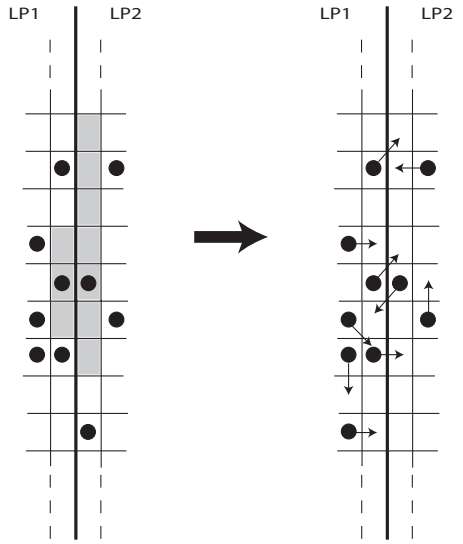


Fig. 6.5. Another scenario of conflicting agents

In Fig. 6.5 a more complex scenario is portrayed. In this example it is assumed that the action radius is set to 1. The cells highlighted are the cells upon which conflicts by agents can occur. In the right side of the figure it is shown a possible consistent advancement of the simulation (i.e. the agents have moved without causing conflicts) where different agents moved at the same time. Fig. 6.6 shows the portion of space that is shared by agents in the case the action radius is equals to 2.

Potentially conflicting agents are characterized as having intersection between their spheres of actions, i.e. the spheres determined by the action radiuses. It is easy to see that the spheres of action of two agents have an intersection when the two agents are far each other less than $2 * actionradius$. In order to prevent conflicts while favoring parallelism among the agents, each border agent is tagged with a *Collision-Free Number (CFN)* that satisfies the following *conflict-free assumption*: two agents which are distant less than $2 * actionradius$ and belong to different LPs must be flagged with a different

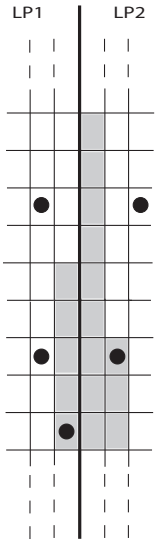


Fig. 6.6. An example of conflicting agents when action radius is set to 2

CFN. The conflict-free assumption ensures that potentially conflicting agents will have different CFNs.

CFN is used to define a partial order relation among agents which prevent conflicts.

CFNs are assigned to agents on the basis of their spatial positions. CFNs assignment can follow a repetitive pattern in which the conflict-free assumption is met. For example, in Fig. 6.7, is shown a repetitive pattern in the case the action radius is 2. The cells flagged with CFN equals to 3 can use the same CFN because agents located on them can not conflict each other. In these cells, indeed, can not exist agents which are both belonging to the same LP and are far from each other less than or equal to $2 * actionradius$. The pattern gets shuffled from time to time, to avoid "privileged" positions. This is achieved by using a pseudo-random generator which takes the virtual time as **seed**. The latter property ensures that the assignment algorithm, played at both sides of a boundary region and at a same virtual time, despite shuffling, remains deterministic: each adjacent LP takes the same decision about the CFN values of shared positions. It should be noted in Fig. 6.7 that the partial-order re-uses as most as possible the same CFNs. CFN reuse is important because it improves the concurrency degree of agents.

6.3 Using time as a tie-breaking mechanism

In discrete event simulation [Fuj00] two different notions of time exist: *logical time* and *wall clock time*. Logical time (or virtual time) is the time related to

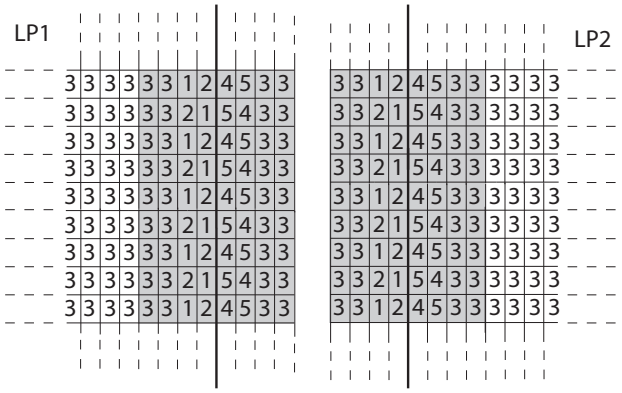


Fig. 6.7. A CFN repetitive pattern

the model, namely the time advancement of the simulation. Wall clock time refers, instead, to the real time during the execution of the simulation itself. Logical time and wall clock time are not tied together except in the case of a real time simulation. *Time management* is in charge to keeping a coherent virtual-time advancement among LPs.

In the approach proposed in this chapter the time management is based on a notion of a *composite logical time* (CLT), i.e. an expanded version of the logical time. CLT is constituted by the virtual-time plus other information (fields) useful for conflict resolution and consistency issues. The use of CLT is based on the assumption that virtual-time only defines a partial order relation among the execution of events. Events marked with the same virtual-time could be potentially executed in any order. CLT enforces a particular execution order among events so as to guarantee conflict resolution and data consistency during the simulation.

6.3.1 The basic version of CLT

CLT is a number which in binary can be seen as the concatenation of two slots: the virtual time in the most significant slot and the *Collision-Free Number* (CFN) (see section 6.2.2) in the least significant one. It is worth noting that the model remains unaware of CFN, whereas the entire CLT constitutes the "logical time" from the time management point of view. The basic structure of the adopted CLT is thus:

```
< -------CLT-------- >
[ Virtual time | CFN ]
```

Given an event scheduled to be received by an agent at a certain virtual time t, the time-stamp of the event is set to the composite logical time where the slot of virtual time is set to t and the CFN is computed as described in

the previous section. As a consequence, a *tie-breaking* mechanism is ensured among events scheduled at the same virtual time, which always guaratees that potentially conflicting agents will be processed at different wall clock times.

6.3.2 Consistency among updates: adding the *step* slot

As stated before, every change occurring in a local border part must be reflected in the mirror part of an adjacent LP. All the changes occurred in a local border of an LP are gathered in a single update message and sent toward the LPs containing the relative mirror parts. In order to avoid the processing of "stale" data, it must be ensured that update messages have to be actualized before the agents, potentially influenced by such updates, undertake a new action. To prevent this, a further *step* slot is added to the CLT thus:

```
<  ---------CLT-----------  >
[ Virtual time | CFN | Step ]
```

Step field is set to 1 e.g. for the update messages. In other cases the field is kept to 0. More in particular, given a state change occurring at a specific virtual-time and CFN, the time-stamp of an update message will be marked with a CLT having the same virtual time and CFN but the step slot is set to 1.

6.3.3 Multiple events at the same virtual-time: adding the *epoch* slot

The notion of CLT as described before is not enough for ensuring a coherent and consistent model evolution in the general case. For example, let us consider a scenario in which the action radius of agents is set to 1 and in which the agents move from cell to cell by receiving a *tick* event. If two tick events are scheduled to be received by an agent at the same CLT, the agent behaves as it would move of two cells at the same time, thus violating its action radius. In order to avoid such a problem, it is required that only one event must be processed by an agent at the same CLT. Another slot named *epoch* is purposely added to the CLT for dealing with this problem. If an agent has already processed events at a given CLT, the next event at the same CLT will be rescheduled at a next epoch. The definitive structure of CLT is portrayed below:

```
<  --------------CLT-------------  >
[ Virtual time| Epoch | CFN | Step ]
```

Another case in which the epoch is used is when an agent sends an immediate event (i.e. at the same virtual-time) to an agent marked with a lesser CFN. To avoid scheduling events in the past, the epoch slot is incremented again.

6.3.4 Remote operations

Read and write operations occurring respectively out of the visibility radius and action radius are referred to as remote operations. Extending the above described approach by considering all the territory as "neighbourhood" would result in the use of a huge number of different CFNs limiting the parallelism within the entire system. For this reason remote operations will be administered asynchronously through the use of suitable *read* and *write* messages exchange between agents and the environment.

As previously described in section 6.2, the asynchronous approach needs to be assisted by further mechanisms offering read/write operations to be executed atomically in order to prevent data inconsistency.

In the approach proposed in this chapter, the atomicity is guaranteed without resorting to an explicit locking mechanism. The CLT is used again for that purpose.

Messages exchanged between agents and the environment are scheduled with a CLT in which CFN is set to a specific reserved value in order to avoid conflict between remote operations and operations in the neighbourhood. If multiple agents refer to the same remote cell at the same virtual-time, a different (incrementally) value of the epoch is used to ensure atomicity among remote operations. In particular, a specific epoch value is assigned to any involved agent. Given an agent, the assigned epoch remains the same for all the operations concerning the same cell at the same virtual-time.

6.4 Supplying *stage* to actors in Theatres

In the following the mechanisms described in sections 6.2 and 6.3 are used for implementing a *stage* concept (i.e. a spatial environment) as first class entity for the Theatre multi-agent architecture [CFN09]. In the following a conservative synchronization structure [Fuj00] [CFN09] is assumed which is based on HLA and is capable of handling actor migrations.

A stage for actors was achieved by providing a first class entity for managing spatial environments in the Theatre architecture. A special actor (`ActorEnv`) is introduced per LP which is devoted to implement the mechanisms previously described in this chapter. A suitable model interface (API) is provided through which the agents can access/modify the territory. In addition, the control machine was customized to host the composite logical time notion illustrated in the previous section.

Actors interact with the `ActorEnv` for both local (i.e. in the neighbourhood) or remote read/write operations. Read/write operations concern the shared state of the environment. This shared state is composed by the state of each cell (i.e. by the actors located on it) along with the shared state of the agents. An agent can declare public-accessible variables (shared variables) through the environment. It is worth noting that other approaches to sharing

agent state could be exploited, e.g. the agents could provide setter and getter methods. In this way, though, supporting the mechanisms proposed in this chapter, would require the use of techniques like aspect oriented programming [KHH+01] or Java reflection.

Remote operations are concretely implemented by message exchanges among actors and the `ActorEnv`. The message interface of the `ActorEnv` foresees four kind of messages:

- *Read*: used to request the content of a remote cell of the territory
- *Write*: used to situate or move an agent in a remote cell or change the shared state of a remote agent
- *ContentMsg*: is the reply of a Read message containing the list of the actors located in the read cell along with the relevant shared agent status
- *PutSucceeded*: is the reply to a write message. It simply triggers an agent to continue its behaviour after a write operation.

For the operations 'in the neighbourhood' the `ActorEnv` furnishes a "view" of the neighbourhood of an agent through the following method:

```
Neighbourhood neigh = ActorEnvironment.getMyNeighbourhood();
```

Fig. 6.8. Use of the ActorEnvironment for getting the Neighbourhood

Despite the usually Java programming style, requesting the neighbourhood does not require the specification of the identity of the requesting actor. `ActorEnv` is in charge to retrieved such identity by querying the Theatre runtime infrastructure. This avoid a malicious and guideless use of the `Neighbourhood`. Actors can operate in their neighbourhood by exploiting functionality offered by the `Neighbourhood` interface shown in Fig. 6.9:

The software engineering process underlying the design of the realized framework is also devoted to foster *robustness* so as to limit possible errors during the modelling activity. For example, an actor is prevented to ask its `ActorEnv` for a `Neighourhood` of another actor (see figure 6.8). Moreover, the use of the `Neighbourhood` interface enforces the constraints about visibility and action radiuses.

As one can see, the interface functionalities provided by the API in Fig. ref make the modeller completely unaware of the distribution concerns. Modeller remains in charge to carefully choosing the size of visibility and action radiuses. Using a wrong size for such radiuses could introduce an high synchronization level during the distributed execution. For example, reducing the size of the radiuses increases the number of remote operations, while choosing a big size implies the number of required CFN increases (see section 6.2.2).

`Actor createAndLocate(String actClass, Position p);`
creates a new actor belongs to the `actClass` and locates it to the position `p` and return the just created actor

`void moveActor(Actor act, Position p);`
moves the actor `act` to the new position `p`

`void removeActor(Actor act);`
remove the actor `act` from the territory. After removed `act` results no longer situated

`List<Actor> getCell(Position p, Class actClass);`
return a list of actors contained in the cell with `p` position matching to a given `actClass` class

`boolean isCellEmpty(Position p);`
check if a cell is empty

`List<Actor> getActors(Class actClass);`
return a list of actors contained in the neighbourhood matching with `actClass` class. The returned list is ordered by distance with respect to actor invoking the method

`Position getPosition(Actor act);`
return the position of the `act` actor

`void addShared(String name, Class type);`
add a shared variable named `name` of class `type` for the actor which call the method

`<T> T getShared(String name, Class<T> type, Actor act);`
return the value of the variable `name` belonging to the `act` actor. For type-safety the class of the variable must be specified

`<T> void setShared(String name, Class<T> type, T value, Actor act);`
change the value of the variable `name` belonging to the `act` actor to the new `value` value. For type-safety the class of the variable must be specified

Fig. 6.9. Neighbourhood interface

The values of these radiuses are estabilished during the configuration phase. In this phase too the partitioning schema of the territory has to be specified.

6.5 Distributing RePast on top of Theatre[2]

RePast (see 2.4.1) is a popular agent toolkit with proven capabilities to fulfill the modelling challenges of large multi-agent systems (MASs). The toolkit, though, is normally used on a standalone workstation and therefore its practical use can be constrained (in space and time) by the limited available computing resources. In this section is described an original approach `HLA_ACTOR_REPAST` aimed to distributing RePast models for high-performance simulation of complex scalable models. Actors bring to RePast agents such features as migration, location-transparent naming, efficient communications, and a control-centric framework. Distribution and time management concerns depend on the IEEE standard HLA middleware (see 2.3.1. In the following it is discussed details of the software engineering process underlying the development of `HLA_ACTOR_REPAST`. The mapping techniques, assisted by Java text annotations and *aspect oriented programming*, try to minimize "code intrusions" in the original model and favor model transparency. Finally, some experimental data are show which witness the good performance results achieved by applying `HLA_ACTOR_REPAST` to a distributed version of a classic MAS benchmark model.

6.5.1 Related work

An experience of distributing sequential RePast on top of the High Level Architecture/Runtime Infrastructure (HLA/RTI) middleware [KDW00] is described in (Minson & Theodoropolous, 2008) and it is referred as `HLA_REPAST`. HLA was chosen because it eases interoperability with existing simulation systems, and promotes model reuse. The implementation directly integrates the RePast mechanisms within the HLA/RTI infrastructure. A distributed RePast simulator is an HLA federation consisting of multiple interacting instances of RePast sub-models. Distributed synchronization depends on conservative synchronization [Fuj00] which favours transparency and backward compatibility with sequential RePast model. The RePast scheduling algorithm was replaced with a new one which constrains local time advancement in a federate, in synchronization with the rest of the federation. Space/environment objects are mapped on the object architecture (Federation Object Model or FOM) and the publish/subscribe design pattern supported by HLA. A critical problem concerns concurrent access/update to shared attributes, e.g. of the environment. Conflicts resolution is achieved by divesting attribute ownership to RTI. All of this can have performance penalties in the runtime. A similar realization was previously experimented with distributing the `SIM_AGENT` toolkit on top of HLA [LLT07]. A different approach is described in [LCZ07] where a federated agent-based architecture for crowd simulations is proposed. Here, a RePast crowd model is used for studying individual and group/environment

[2] based on [CFGN10a, DAAL09, CFGN09a]

behaviours which result from interactions among individuals and the individuals and the environment. The RePast model is mapped onto one HLA federate which is integrated with ontology and inference system for dynamically adjusting the behaviour of individuals and of the environment. The RePast federate can interoperate with other, possibly heterogeneous, special components, e.g. devoted to visualization purposes. Therefore, in this approach the RePast model is not partitioned but only exposed for interactions in a distributed domain. Another approach -`REPAST_JDSM`- to interfacing RePast with the HLA is discussed in [YWCTM09] where a generic architecture supporting Commercial Off-The-Shelf simulation package interoperability is adopted which hides and simplifies the use of HLA services. The approach favours interoperability of RePast models by relying on an explicit sending/receiving entity mechanism. `REPAST_JDSM`, though, has to be extended for dealing with conflict management among shared variables. In [TZC+06][CTT+08] the `HLA_GRID_REPAST` architecture is proposed which combines `HLA_REPAST` concepts [MT08] with `HLA_GRID` platform (Zong et al., 2004) in order to favour cooperative development of simulation services, with automatic discovery and deployment of services in a distributed model, and with the possibility of accessing large and geographically decentralized data sets. The proposal addresses the vision of the Grid as a *plug-and-play distributed simulation system*. Flexibility and openness of the architecture have the consequence of reducing the achievable performance with respect to `HLA_REPAST` which in turn can suffer from the use of HLA object management mechanisms. Another approach to distributing RePast models has been recently proposed by integrating the visual toolkit RePast Symphony with Terracotta (see 2.3.2) object infrastructure. The use of Terracotta enables JVM-level clustering and offers a shared virtual heap. The approach requires time coordination at-large and has still to demonstrate its performance potential.

6.5.2 Inside RePast

The RePast toolkit comes with a *runtime executive* (*scheduler* and *controller* components) which provides an event-driven simulation engine, and a *user interaction interface* through which a simulation experiment can be controlled. Typically, a system (see Fig. 6.10) consists of a collection of *agents*, a collection of spaces modelling the physical environment within which the agents are situated, and a *model* object which contains information (e.g. for configuration) about the entire system. The state of system is scattered among model, agent and space objects.

An agent-based simulation normally proceeds in two stages. The first one is a *setup* stage that prepares the simulation for execution. The second stage is the actual *running* of the simulation. During the setup phase, the model object is created as an instance of a Model class (implementing the SimModel interface) which, in turn, instantiates agents and spaces, the display and the scheduler (the latter is an instance of the Schedule class). It is the executive

which actually asks the model object to execute the setup phase. After that, i.e. when the simulation has been started, the executive achieves from the model object the scheduler object used to control the simulation. Fig. 6.10 sketches the operations of model bootstrap, whereas Fig. 6.11 highlights model execution.

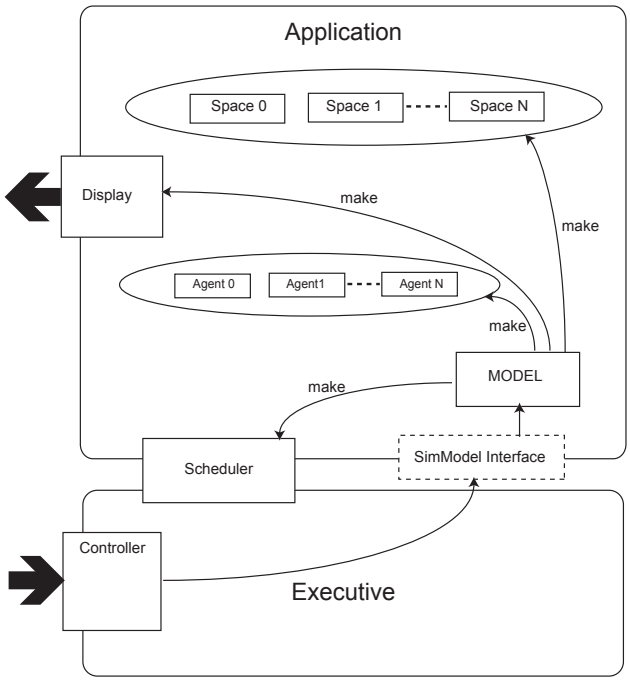


Fig. 6.10. Startup scenario

RePast events are instances of the BasicAction class. An event occurrence is mirrored by an invocation of the execute() method of a basic action object. Actions are scheduled to occur at certain simulation times (*ticks*). Ticks can be equally spaced or, more in general, not equally-spaced or event-driven. All pending events existing at a given moment are buffered, ranked by ascending timestamps, within the scheduler object. At each iteration of its control loop, the controller asks the scheduler to extract the (or one) most imminent pending action and to dispatch it to its destination object, i.e. model or agent. The consequence of an event occurrence is in general a chain of method invocations, which can cause state changes in agents, in model or in space objects.

Space objects can be *data* (or *diffusive*) spaces or *object (agent) spaces*. An application based on diffusive spaces typically has the model which repeatedly executes a cycle made up of three basic phases: *diffuse-modify-update*. In the *diffuse phase*, the environment is queried to synchronously update itself

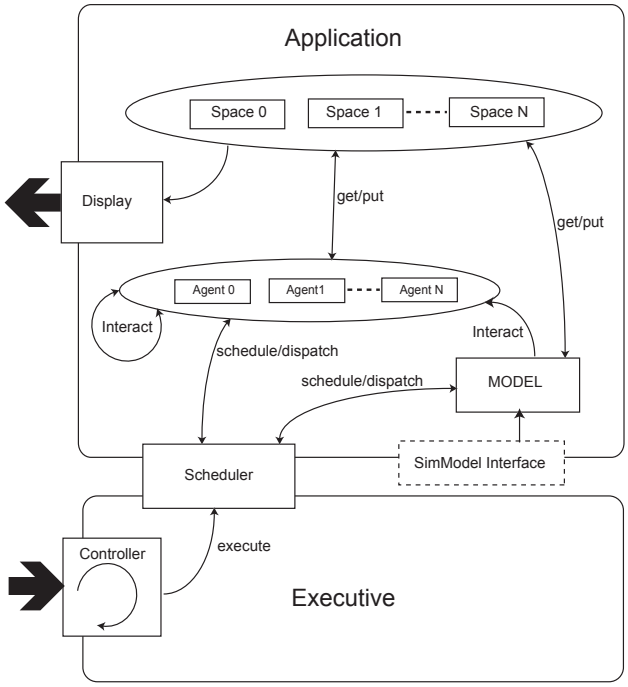


Fig. 6.11. Runtime scenario

according to a diffusive logic. Then, in the *modify phase* agents are permitted to introduce further changes to the data space. For consistency, though, these changes are stored in a temporary copy of the environment. Finally, in the *update phase*, the temporary copy is restored upon the actual environment. An object space, on the other hand, behaves more asynchronously. Agents can issue get/put operations to spaces, which affect immediately the environment. At each tick, the Model object causes the environment changes to be displayed by invoking the Display *redraw* method.

6.5.3 `HLA_ACTOR_REPAST` Design Issues

Distributing sequential RePast multi-agent models requires in general addressing some fundamental design issues concerning: event scheduling/dispatching and time management, state consistency, conflict resolution on shared variables. These issues are also tied to the distribution of the spatial environment (i.e., territory) where the agents are situated.

Different strategies can be adopted for model partitioning [MT08]. A particular strategy may depend both on load balancing issues and on the way shared variables are accessed or modified [DET+08][**?**][VMT09]. Space partitioning adopted as well as conflict resolution and to ensure consistency

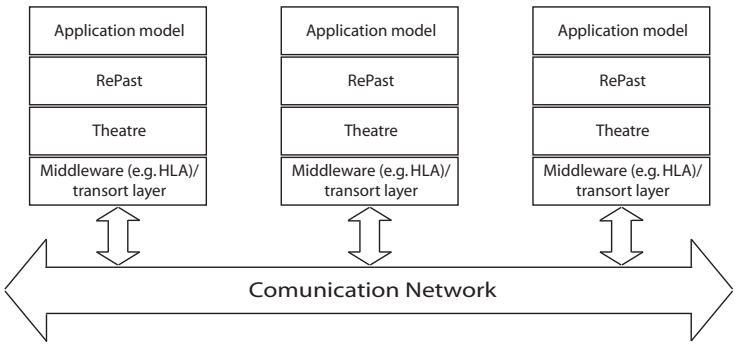


Fig. 6.12. Architecture of a distributed RePast model

on shared state in this work rely on ideas and mechanisms previously described (see 6.2. This because presence of *space objects* qualify RePast as a *situated* agent-based infrastructure. In particular we make the assumption that an agent can read and update only the spatial environment belonging to its surrounding territory (*neighbourhood*). Such neighbourhood represents the *"sphere of influence"* [LT01] of the agent, i.e. the portion of shared state which can influence or that can be influenced by the agent behaviour. More specifically, a RePast model is assumed to be partitioned into a collection of LPs which are allocated for the execution to different computing nodes of a networked system (see Fig.6.12). Each LP/federate hosts a portion (*region*) of the environment and a subset of agents (turned into actors).

Mapping RePast onto actors

Figure 6.13 summarizes the mapping process of RePast over actors. Every RePast agent becomes an actor and its class inherits from a specific actor class `-ACTOR_AGENT-` which is specialized in handling RePast agent concerns. `ACTOR_AGENT`, in turn, inherits from the Actor base class. In the same way, the class of a model object inherits from `ACTOR_MODEL` which derives from Actor base class.

RePast action objects are transformed into actor messages and scheduled in the theatre control machine of the LP. A key point of `HLA_ACTOR_REPAST` is a transfer of control responsibilities from RePast to the theatre control machine. Basically, RePast controller remains in charge of interactive events only. Simulation and time management services are instead provided by theatres. In the proposed mapping, local action executions behave exactly as in the non distributed version of the model. However, remote action requests, which occur when the target agent is located on a remote LP, are converted into actor messages and sent across the network. In a similar way, a method

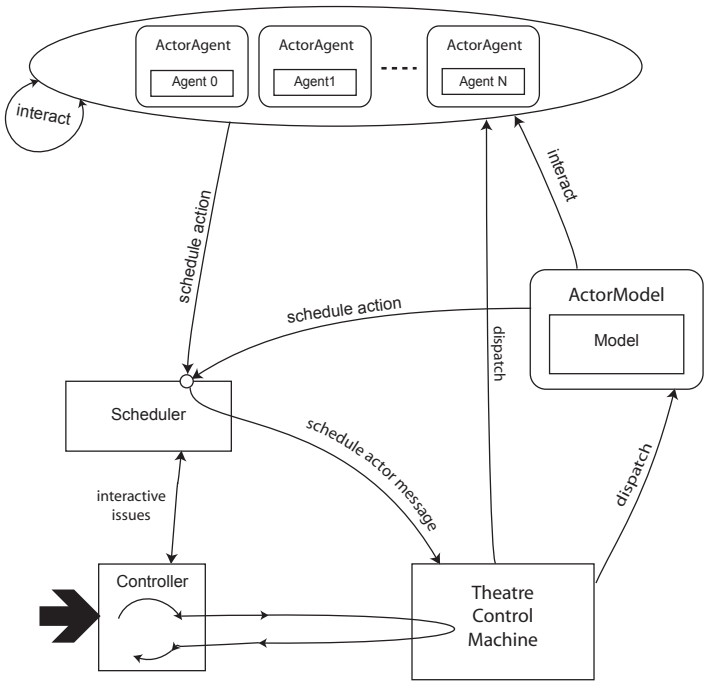


Fig. 6.13. Outline of RePast mapping onto actors

invocation on a remote object is wrapped in a message which is sent to the target object. The actor message carries the method name and specifies the receiver `ACTOR_AGENT` (or `ACTOR_MODEL`) upon which the method has to be invoked. On receiving one such a message an `ACTOR_AGENT` (or `ACTOR_MODEL`) is logically responsible, through its handler() method, of invoking the specific method of the corresponding RePast agent, e.g. with the help of Java reflection. In another case, when a message is up to be dispatched to a forwarder, i.e. a proxy version of an actor, the `ACTOR_AGENT` (or `ACTOR_MODEL`) could manage to route the message to the corresponding remote actor. In reality, dispatching a message to a forwarder is a concern directly handled by the control machine. In addition, the recourse to "heavyweight" reflection techniques is avoided. The handler() method of `ACTOR_AGENT` (or `ACTOR_MODEL`) is customized with application-specific code that directly invokes the agent method. It is worth noting that an `ACTOR_AGENT` stores through instance variables such common data as the agent coordinates and temporal information which are required by the distributed model execution. An important actor, created in every LP during start-up, is the "environment actor" (EnvActor) which knows about configuration information (taken from a configuration file) of the entire RePast model, and offers a common interface for accessing spaces and environments. For example, the provision that every node/LP is an instantiation

of the "entire" model implies that within each sub-model assigned to an LP be present the "global view" to the environment. The transformation of positional coordinates of a situated agent from the global-view to the local-view corresponding to the portion of the environment effectively managed by the LP is realized by the corresponding instance of EnvActor (see Fig.6.14).

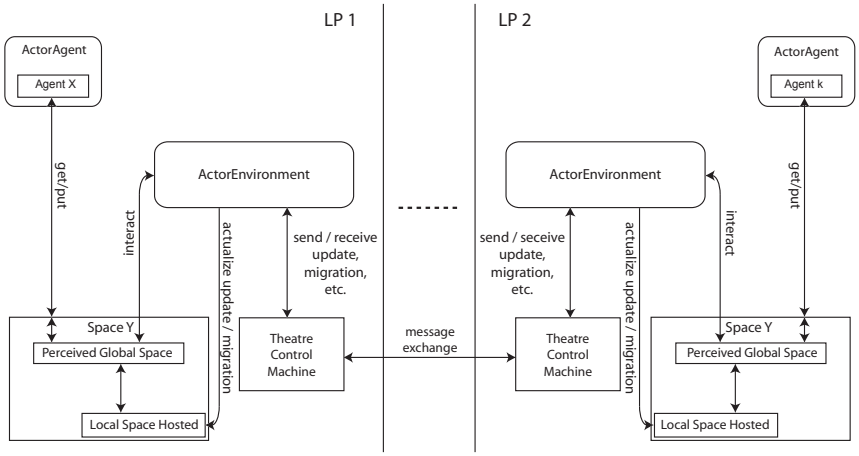


Fig. 6.14. Actor environments and local vs. global environment view

A second responsibility of EnvActor is that of propagating to its peer(s) in neighbouring LP(s) the updates to agents in the local border of the belonging LP. Similarly, when an agent moves to the border region, the EnvActor is in charge of migrating the agent, if needed, to an adjacent LP. EnvActor also plays a key role in the diffuse/update/redraw processes. In particular, when the model raises a diffuse/update/redraw operation, the request is intercepted and a corresponding message is made and sent to all the LPs, which is finally heard by the EnvActors. An EnvActor then acts so as to actuate the diffuse/update/redraw operation in the local environment.

Text annotations and aspects

The software engineering process underlying `HLA_ACTOR_REPAST` was driven by the main goal of keeping transparency between the sequential model and its corresponding distributed version. The mapping techniques described in previous subsection, indeed, are realized without changing code in the source model (except for a few text annotations described in the following). This goal is achieved through the use of aspect oriented programming, in particular through an AspectJ *aspect* [KHH+01][Asp]. The Aspects supply mechanisms to transparently change source code in a static way, through the so

called *introductions*, and in a dynamic way (at runtime) through the so called *interceptors*.

Two kinds of static "introductions" are needed to make the model compliant with `HLA_ACTOR_REPAST`. The first one concerns the requirement that each RePast agent class has to inherit from the `ACTOR_AGENT` class. The second kind of introduction affects specifically the handler() method of agents. The introduced handler() method is filled with application-specific code for directly invoking an agent method in response to a received message.

A first type of interceptor is triggered on the occurrence of method invocations on an agent or a model object. The aspect interceptor captures method calls and superimpose to them a suitable behaviour. When the target agent of a method invocation is local, the aspect lets the agent execute the method call in the normal way. In the case the receiver is remote, the aspect replaces the standard behaviour of method call by building a message, filling it with information about the requested method, and sending the message over the network (see Fig. 6.15). This interceptor and the introduction which regards the handler() method make `ACTOR_AGENT` independent of any particular model.

A second kind of interceptor is responsible of ensuring agent state consistency. When an agent is located in the border area and a state change occurs in it, the change must be reflected in adjacent LP(s). Therefore, when a change refers to an application relevant field of an agent, the aspect interceptor captures the field setting and acts to achieve a suitable behaviour: in the case the agent is on a border the aspect asks the EnvActor to notify the update to the appropriate neighbour LPs; otherwise, the aspect annotates the field change in the actor agent part of the agent (see Fig. 6.13). As soon as the agent moves to a border position, the EnvActor will be in charge of updating the mirror version of the agent on adjacent LPs thus avoiding them to refer to a stale status.

Application specific information are required to build the above-described Aspect and link it to the model. The information are supplied by the modeller through a few Java text annotations. An annotation processor was developed which reads these annotations in the source model and automatically generates and customize the Aspect code.

The required annotations are `@Agent`, `@Method` and `@Field`:

- `@Agent` qualifies a class as a RePast agent class
- `@Method` qualifies methods of RePast object classes (of agents and model) which will be called on behalf of another object of the model. Such methods are assumed to return no data (void return type)
- `@Field` qualifies a field of an agent as one which is "relevant" to the rest of the system, i.e. of which consistency update and conflict resolution have to be guaranteed

An excerpt of an annotated class is shown in the following.

```
@Agent
```

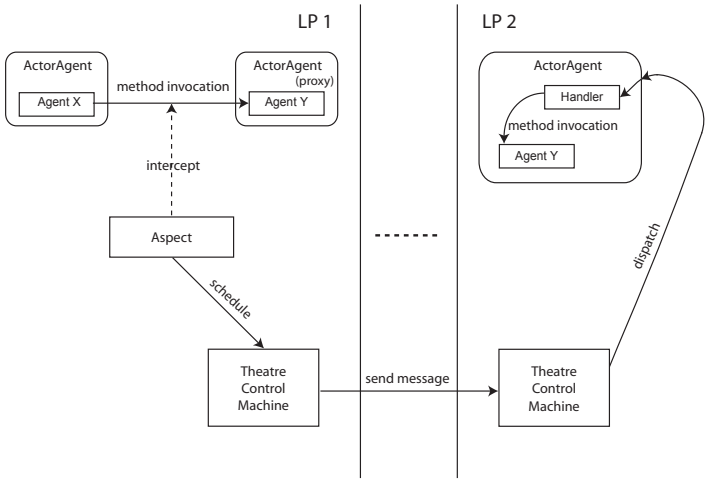


Fig. 6.15. Method invocations and aspect-oriented programming

```
public class AgentClass {
public boolean f2; //field unaffected by consistency updates
@Field public int f1; //field involved in consistency updates

public int statusReading(){} //method reserved for reading the agent status
@Method public void doSomething(){} //method which may be invoked by other agents

}//AgentClass
```

Consistency updates and conflict resolution

Consistency update and conflict resolution in `HLA_ACTOR_REPAST` rest on the approach described in 6.2.2 and 6.3. Composite logical time is used as a *tie-breaking* mechanism. Nevertheless, the approach was specialized to specifically deal with issues related to `HLA_ACTOR_REPAST`.

A main difference relies on inter-agent method calls. In the general approach previously described an agent can synchronously perceive the environment in its neighbourhood as well as the neighbour agent states via the environment but it can not make a method call upon another agent.

The management of inter-agent method calls which RePast makes it possible, was based on Aspect Oriented Programming and Java Annotations as described in 6.5.3. In addition the technique to handle conflicts and data consistency was refined as well.

The state of an agent can change when an action is delivered to it or when a method call is executed on it. Of course, actions scheduled at different

virtual times always ensure causal order. In the case of method calls, however, it must be considered that a method call is not immediately actualized when the target agent is situated on a border. One such a method call is intercepted by the Aspect (see Fig. 6.15) and if the agent belongs to a different LP or if the agent is on a border (which means it has to receive messages with a generation greater than zero) a message must be created and scheduled. As a consequence, an agent state change could be deferred at a future real time (greater generation) although the same virtual time is involved. Visibility problems arise if one tries to read and use the agent state in the meantime. This is a second kind of conflicts which can occur in the distributed version of a RePast model. The following proposes a refinement of the partial-order described in section 6.3 which in its basic form is not capable of dealing with the new kind of conflicts too. In particular the *Collision-Free Number (CFN)* needs to be refined.

To clarify problems, consider a scenario with three agents, two of which are in charge of shooting the third one modelling a target. When an agent shoots the target, the latter could decrease its energy (or vitality degree) so that when the energy becomes zero agents can stop shooting it. Shooting can be modelled by a method call upon the target agent. Suppose now that, at a given virtual time, the three agents are located in three contiguous border cells, and that cells containing the shooting agents have respectively 1 and 2 as the assigned CFN, whereas the cell containing the target agent has 3 as the assigned CFN. In the hypothesis that both shooting agents try to shoot target at the same virtual time, the agent in the cell tagged with CFO 1 will be the first executing and shoots the target. Since agents are in a border area, the method call is intercepted and a corresponding message created and scheduled at current virtual time but with the CFN dictated by the cell hosting the receiver, in this case the value 3. Next, it is the turn of the agent in the cell tagged by CFN value 2. The latter agent considers target vitality in order to decide if shooting the target or not, without any knowledge of the fact that the target has already been shot, this because the target agent has not yet received the shooting message which decreases its vitality. The solution proposed for coping with the above problems splits the CFN field in the composite logical time, in such a way that the CFN consists of multiple sub-fields. In the discussed example it is important that the target receives the shooting message *before* the CFN becomes 2. Roughly speaking this means that the message could be scheduled with a CFN equals to 1.x. In this way, the target agent would decrease its vitality at composite logical time with CFN 1.x and step 0 although the change would be reflected in neighbour LP actually at CFN 1.x and step 1 due to the management of update messages. As a consequence, the second agent running at composite logical time with CFN 1.x and step 0 will be able to correctly read and see the updated vitality status of the target regardless the LP which hosts it. More concretely, a "composed" CFN looks like gs.gr where gs is the CFN assigned to the cell hosting the caller agent and gr is the CFN assigned to the cell holding the called agent. This

provision, paired with space cell conflict resolution mechanism 6.2.2, ensures that method calls on contiguous border agents, raised at a same virtual time, are always cell conflict-free because they will necessarily occur at different real times. In general, a chain of method calls can occur. If an agent receives a message with CFN x.y and reacts by calling a method on another border agent, the corresponding message would be scheduled with generation x.yz where z is the CFN assigned to the cell of the called agent. The solution demands for a flexible fine-grain CFN field, i.e. with a certain number of sub-fields ranging from a most significant one down to a least significant one. Actually, the `HLA_ACTOR_REPAST` allows the modeller to configure the composed logical time by specifying the number of bits allocated to the virtual time and that allocated to the CFN (the step field is always one bit long). Moreover, the internal decomposition of the CFN into sub-fields can be customized according to the estimated maximum nesting level of a method-call chain. The control machine, Env_Actor and the aspect/interceptor are in charge of handling the described refinement of the partial-order, transparently with respect to the source model.

6.5.4 Tileworld Model Example

Tileworld [PR90] is a widely used benchmark for testing multi-agent systems. It defines an environment with tiles, holes and obstacles. Agents are able to move over the environment in order to pick up tiles and subsequently using them to fill holes. Tiles and holes appear and disappear randomly. When a tile is picked up by an agent it disappears definitively, when a hole is totally filled it disappears definitively as well. Each hole has an associated value. An agent knows ahead of time the value associated with the visible holes. An agent gets score when totally filling a hole. The goal of each agent is to maximize its score as possible. A representation of the game is portrayed in Fig. 6.16. The model developed in this paper differs from that proposed in [PR90] in two points:

- holes size is one cell but each hole owns a 'depth' that is the number of tiles that the hole can contain
- agents can explore the territory moving also in diagonal directions.

The behaviour of an agent can be modelled according to two alternative strategies: reactive and deliberative. In the reactive strategy the agent follows the behaviour recapitulated by the statechart model depicted in Fig. 6.17. At its start-up, the agent is in the `LOOK_FOR_TILE` state where it checks its neighbourhood in order to detect visible tiles whilst avoiding surrounding obstacles. A tile is visible if it falls in the visibility area of the agent. In this state, other agents and holes are considered obstacles as well. On detecting a tile, the agent switches to the `MOVE_TO_TILE` state and heads towards reaching the tile. When reached, the tile is picked up and the agent moves into `LOOK_FOR_HOLE` state. While moving to tile, if the latter disappears the agent comes back into

LOOK_FOR_TILE state. The behaviour of the agent into LOOK_FOR_HOLE and MOVE_TO_HOLE states is similar to the behaviour exhibited respectively when looking for a tile and moving to a tile. In this strategy agent does not take care about a value of a hole. The deliberative strategy is borrowed from Pollack & Ringuette strategy: each agent plans its behaviour on the basis of an appropriate metric. It reconsiders its plan using only a filtered set of environmental information (more details in [PR90]). When an agent is in LOOK_FOR_TILE or MOVE_TO_TILE states it behaves as in the other strategy. However, the choice of the hole is made by considering the visible holes and computing the SEU indicator (Subjective Expected Utility):

$$SEU(h) = \frac{score(h)}{dist(a,h) + tileavail(h)} \tag{6.1}$$

where a is the agent, h is the hole, $score(h)$ is the value of h, $dist(x,y)$ is the cell distance between x and y in the territory and $tileavail(h)$ is defined as:

$$tailavail(h) = \sum_{i=1}^{n} 2 * dist(t_i, h) \tag{6.2}$$

where n is the depth of h and t_i is the i_{th} nearest tile with respect to h. In practice, $tileavail(h)$ estimates the travelling needed by an agent in order of totally filling the hole h.

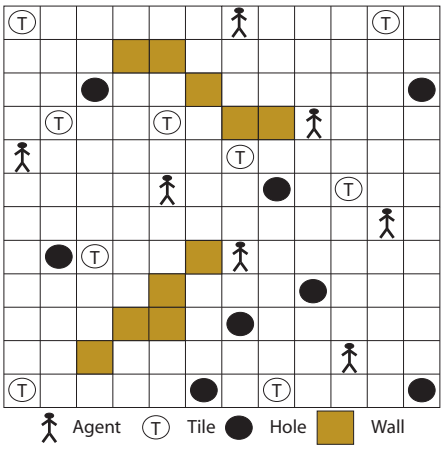


Fig. 6.16. A Tileworld board

A RePast model of Tileworld consists of the following entities:

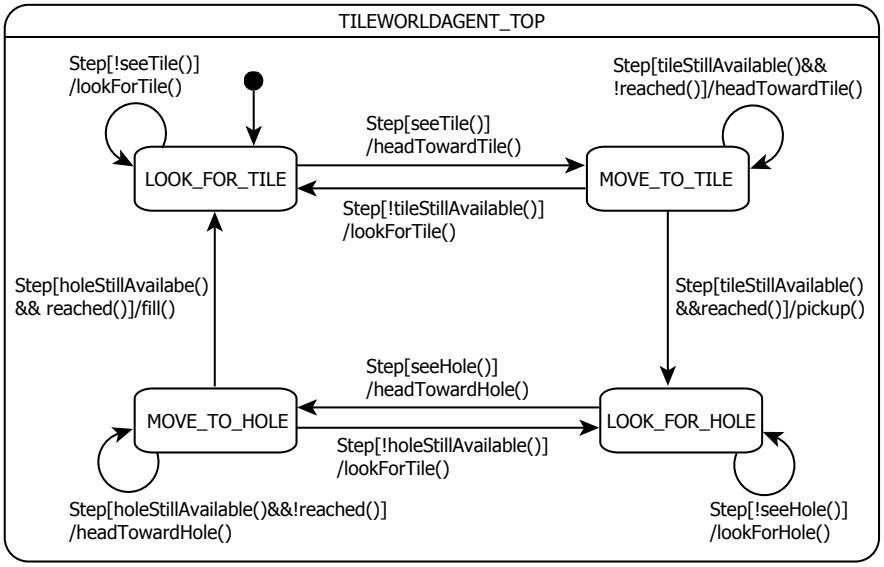


Fig. 6.17. TileWorldAgent behaviour

- (i) TileWorldAgent objects modelling the agents moving around the environment. The behaviour of these agents is triggered by invoking, at each simulation step, a step() method
- (ii) a TileWorldModel object which is in charge of configuration and deployment of the entire system and the scheduling of an action to be executed, on itself, at each tick. Each time the model object receives this action, it invokes the step() method on each TileWorldAgent
- (iii) Tile objects
- (iv) Hole objects
- (v) Obstacle objects
- (vi) a bi-dimensional toroidal grid modelling the spatial environment. Each cell in the grid can contain one single object.

Environment configuration is assisted by a visual editor. A randomly generated environment can be also obtained by simply specifying the environment dimension and the number of entities to put on it.

6.5.5 Simulation experiments

The Tileworld model based on the deliberative strategy was used as a test-bed for checking the simulation performance achievable with `HLA_ACTOR_REPAST`. A set of simulation experiments was carried out where the model consists of a fixed large territory of 825x276 cells and a variable number of agents, tiles and holes, randomly initialized on top of the environment. The model was

equally split into three LPs/federates allocated on three Pentium IV, 2 GHz, 256MB RAM, WinXP platforms, hosting RePastJ 3.0 and interconnected by a Gigabit Ethernet switch, in the presence of HLA pRTI 1516 (Pitch, online). Each theatre/federate hosts a sub model which manages a territory of 275x276 cells. The time required by an agent to make a move is 1 time unit (tu). The dwell time of a tile/hole into a cell is uniformly distributed in [1..50]. The re-appearance time for a tile/hole is uniformly distributed in [1..20]. The simulation time limit was then chosen by ensuring that with the above time parameters, agents would "complete their mission", i.e. picking-up and dropping into holes almost all tiles, by the end of simulation. In particular, a simulation time limit of 200 tu emerged. Some preliminary tests used the same model executed separately using RePast and `HLA_ACTOR_REPAST`. Differences introduced by the use of `HLA_ACTOR_REPAST` concern the addition of the appropriated Java annotations as described in section 4.3, and the use of a configuration file containing distribution information such as IP & port of the various LPs, visibility radius of each agent i.e. border size etc. Wall clock time was measured in both cases and the ratio between RePast wall clock time and `HLA_ACTOR_REPAST` wall clock time was calculated for performance comparison. Experiments used a configuration of 500 Agents, 250 holes, 250 tiles and 100 obstacles randomly spread across the territory. This configuration was scaled by a K factor which ranges from 1 to 10 in order to test speed-up vs. an increasing load. Fig. 6.18 summarizes the measured speed-up vs. the number of agents in these preliminary tests.

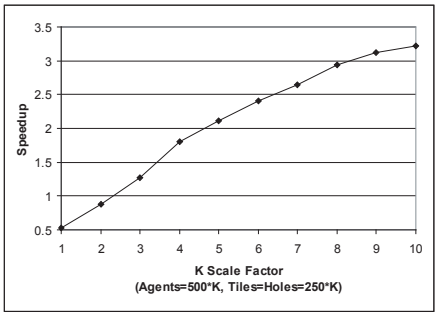


Fig. 6.18. Preliminary tests showing super speed-up (three processors)

As one can seen from Fig. 6.18 a super speed-up emerged. Another set of experiments were carried out specifically for taking a comparison between the model ran by sequential RePast and the model executed using `ACTOR_REPAST`, i.e. the system developed in this work but running on a standalone machine. The adopted territory configuration was the same but K now ranges from 1 to 5. Fig. 6.19 shows the observed wall clock time in both cases. Reasons underlying the behaviour depicted in Figures 6.18 and 6.19 are concerned with

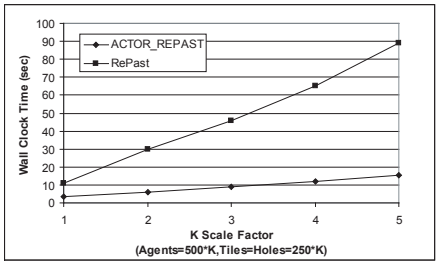


Fig. 6.19. Comparison of sequential RePast vs. sequential `ACTOR_REPAST`

creation/scheduling of actions. There are many scheduling methods offered by RePast. However they can be roughly divided in two method types: one that accepts an action already created by the model itself and another method that accepts reference to the called object and name of the method to call. This latter scheduling method was used in the Tileworld model. When a model schedules an action by supplying object reference and method name, RePast dynamically creates the bytecode of a new action class, makes an instance of this class and finally schedules it. `ACTOR_REPAST`, instead, always directly creates and schedules an instance of a specific existing message class, so no creation of bytecode is ever needed. For subsequent experiments, it was decided to compare the performance achievable by `ACTOR_REPAST` (centralized) and by `HLA_ACTOR_REPAST` (distributed), by varying the number of agents from 5000 to 52500 (as permitted by memory constraints of utilized machines). Alternatively, the original RePast model could have been modified so as to use model specific action classes only, thus avoiding the above described overheads. Fig. 6.20 shows the observed wall clock time, while Fig. 6.21 portrays the achieved simulation speedup. From Fig. 6.20 it follows that the model scales well as the number of agents increases.

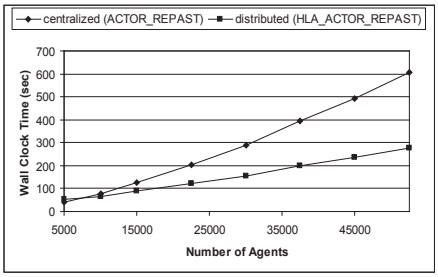


Fig. 6.20. Wall clock time vs. number of agents

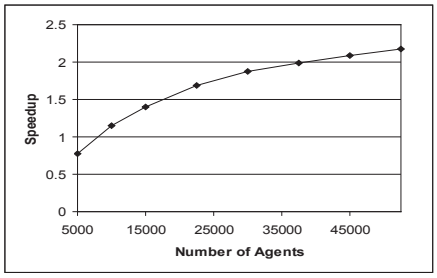


Fig. 6.21. Simulation speedup vs. number of agents (three processors)

Conclusions and Outlook

The work behind this PHD thesis was devoted to designing and implementing *agent-based* tools and approaches, addressing modelling and simulation (M&S) of complex dynamic systems. The starting point was the use of the open, flexible and efficient `Theatre` architecture in Java, developed within the Software Engineering Laboratory at DEIS/UNICAL. Theatre offers an agent-infrastructure centred on a light-weight actor computational model, whose control engine can be customized through programming. Theatre admits a fine-grain *migration* mechanism for actors, and can easily be adapted to work with different transport layers like Java Sockets or Java RMI. In this thesis such standard or emerging middleware were favored as *HLA/RTI* and *Terracotta*. Terracotta, in particular, was appealing for its ability to transparently clustering the JVM thus allowing to experiment with high-performance *parallel/distributed* simulation of scalable systems on top of modern cost-effective multi-core clusters. Theatre was successfully exploited for supporting rigorous modelling languages and formalisms like *DEVS* and *statecharts*. It is felt useful to prosecute the accomplished work e.g. in the following directions.

- Experimenting with a notion of statecharts based actors also in the presence of the *and-decomposition* construct, using the UML semantics which rests on the step notion comprising the run-to-completion processing of one event at a time. Such a semantics is already at the hearth of the adopted Theatre actor model.
- Developing a visual tool in Java supporting the graphical design of an actor model, where actors can have a statechart-based behavior. The tool would allow checking/debugging and code generation of a graphical model, also considering the partitioning task over a parallel/distributed context.
- Optimizing the software engineering project carried out in distributing the sequential RePast toolkit, using it for for high-performance simulation of large variable structure systems on multi-core clusters.
- Improving the management of *spatial environments* in *situated* multi-agent systems. This is a challenging and hot research line in current technical

literature, because of its relevance in many multi-agent systems. The goal is to go beyond the prototype realization underlying the approach to distributing RePast models on top of Theatre actors, and to check it with significant application models and related execution performances.

References

ADE. ADEVS ONLINE. http://www.ornl.gov/~1qn/adevs/index.html.

Agh86. G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

Asp. AspectJ.

Ast99. M. Astley. *The Actor Foundry.* University of Illinois, http://yangtze.cs.uiuc.edu/foundry/foundry.ps, 1999.

BBCM98. M. Breugst, I. Busse, S. Covaci, and T. Magedanz. Grasshopper: A mobile agent platform for IN based service environments. In *Proceedings of the IEEE Intelligent Networks Workshop*, pages 279–290, May 1998.

BGFL94. S. Bandinelli, C. Ghezzi, A. Fuggetta, and L. Lavazza. Spade: An environment for software process analysis, design, and enactment. In *Software Process Modeling and Technology*, pages 223–248. Wiley, 1994.

BHJ+10. Ketan Bahulkar, Nicole Hofmann, Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Performance evaluation of pdes on multi-core clusters. *Distributed Simulation and Real Time Applications, IEEE/ACM International Symposium on*, 0:131–140, 2010.

BHR+02. J. Baumann, F. Hohl, K. Rothermel, M. Strasser, and W. Theilmann. MOLE: A mobile agent system. *Software - Practice and Experience*, 32(6):575–603, 2002.

BHRS98. J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole - concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, 1998.

BIP88. M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

BMS02. S. Bandini, S. Manzoni, and C. Simone. Dealing with space in multiagent systems: a model for situated MAS. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1183–1190, New York, NY, USA, July 2002. ACM Press.

BNO03. R. Beraldi, L. Nigro, and A. Orlando. Temporal Uncertainty Time Warp: an implementation based on Java and ActorFoundry. *Simulation*, 79(10):581–597, 2003.

BR01. F. Bellifemine and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice & Experience*, 31(2):103–128, 2001.

Bra97. J.M. Bradshaw. *Software agents*, chapter An Introduction to Software Agents. MIT Press, Cambridge, MA, USA, 1997.

BRJ99. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

Bro86. R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA–2(1):14–23, March 1986.

Bro90. R.A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1–2):3–15, June 1990.

Bro91a. R.A. Brooks. Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 569–595, Sydney, Australia, 1991.

Bro91b. R.A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, 1991.

BS92. B. Burmeister and K. Sundermeyer. Cooperative problem solving guided by intentions and perception. In E. Werner and Y. Demazeau, editors, *Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 77–92, Amsterdam, The Netherlands, 1992. Elsevier Science Publishers B.V.

BSdS+05. A. Brabazon, A. Silva, T. Ferra de Sousa, M. O'Neill, R. Matthews, and E. Costa. Investigating strategic inertia using OrgSwarm. *Informatica*, 29(2):125–136, 2005.

CDP. CD++ ONLINE. http://cell-devs.sce.carleton.ca/mediawiki/index.php/Main_Page.

CFGN08. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Net centric modelling and simulation using actordevs. In *Proc. of 20th European Modeling and Simulation Symposium (EMSS'08)*, Campora San Giovanni, Italy, 17–19 September 2008.

CFGN09a. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Distributing RePast simulations using actors. In *Proc. of 23rd European Conference on Modelling and Simulation (ECMS'09)*, pages 226–232, Madrid, Spain, June 9-12 2009.

CFGN09b. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Statechart-based actors for modeling and distributed simulation of complex multi-agent systems. In *Proc. of 23rd European Conference on Modeling and Simulation (ECMS'2009)*, Madrid, Spain, 9–12 June 2009.

CFGN10a. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. HLA_ACTOR_REPAST: An approach to distributing RePast models for high-performance simulations. *Simulat. Modell. Pract. Theory*, 2010. doi:10.1016/j.simpat.2010.06.013.

CFGN10b. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Parallel simulation of multi-agent systems using terracotta. In *Proc. of 14th ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT2010)*, Fairfax, Virginia USA, 17–20 October 2010.

CFGN10c. F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Performance of a multi-agent system over a multi-core cluster managed by terracotta. In *submitted*, 2010.

CFN06. F. Cicirelli, A. Furfaro, and L. Nigro. A distributed agent-based simulation model for large wireless sensor networks. In *Proc. of Agent-Directed Simulation (ADS'06)*, pages 115–122, 2006.

CFN07a. F. Cicirelli, A. Furfaro, and L. Nigro. Distributed simulation of modular time Petri nets: an approach and a case study exploiting temporal uncertainty. *Real-Time Systems*, 35(2):153–179, 2007.

CFN07b. Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Exploiting agents for modelling and simulation of coverage control protocols in large sensor networks. *Journal of Systems and Software*, 80(11):1817–1832, 2007.

CFN08. F. Cicirelli, A. Furfaro, and L. Nigro. Actor-based simulation of PDEVS systems over HLA. In *Proc. 41st Annual Simulation Symposium (ANSS'08)*, pages 229–236, 2008.

CFN09. F. Cicirelli, A. Furfaro, and L. Nigro. An agent infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. *SIMULATION, Trans. of SCS*, 85(1):17–32, 2009.

CFNP07. F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo. A component-based architecture for modelling and simulation of adaptive complex systems. In *Proc. of 21th European Conference on Modelling and Simulation (ECMS 2007)*, pages 156–163, 2007.

CGN10. F. Cicirelli, A. Giordano, and L. Nigro. Distributed simulation of situated multi-agent systems. In *submitted*, 2010.

CTT+08. Dan Chen, Georgios K. Theodoropoulos, Stephen J. Turner, Wentong Cai, Robert Minson, and Yi Zhang. Large scale agent-based simulation on the grid. *Future Gener. Comput. Syst.*, 24:658–671, July 2008.

CYLT05. W. Cai, Z. Yuan, M.Y.H. Low, and S.J. Turner. Federate migration in HLA-based simulation. *Future Generation Computer Systems*, 21(1):87–95, 2005.

CZ97. D. Challet and Y.-C. Zhang. Emergence of cooperation and organization in an evolutionary game. *Physica A*, 246(3–4):407–418, 1997.

DAAL07. Cicirelli F. D., Furfaro A., Giordano A., and Nigro L. An agent infrastructure for distributed simulations over hla and a case study using unmanned aerial vehicles. In *40th Annual Simulation Symposium (ANSS'07)*, Norfolk, VA, USA, March 26 - 28 2007.

DAAL09. Cicirelli F. D., Furfaro A., Giordano A., and Nigro L. Distributed simulation of repast models over hla/actors. In *13th ACM International Symposium on Distributed Simulation and Real Time Applications (DSRT'09)*, Singapore, 25-28 October 2009.

DAL08. Cicirelli F. D., Furfaro A., and Nigro L. Actor-based simulation of pdevs systems over hla. In *Anss'08*, Ottawa, ON, Canada, April 14-16 2008.

DET+08. Chen D., R. Ewald, G. Theodoropoulos, R. Minson, T. Oguara, M. Lees, B. Logan, and A. Uhrmacher. Data access in distributed

simulation of complex systems. *Journal of Systems and Software*, 81:2345–2360, July 2008.

DLG+04. M. D'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Sytems*, 9(1–2):5–53, 2004.

DM98. B.L. Danny and O. Mitsuru. *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

EFB01. T. Elrad, R.E. Filman, and A. Bader. Aspect Oriented Programming: Introduction. *Communications of the ACM*, 44(10):29–99, April 2001.

FD02. A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *Proceedings of the First international Joint Conference on Autonomous Agents and Multiagent Systems*, pages 737–744, New York, NY, USA, 2002. ACM Press.

FDB02. J.B. Filippi, M. Delhom, and F. Bernardi. The jdevs modelling and simulation environment, 2002.

Fer99. J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, 1999.

FG97. S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer-Verlag, 1997.

FIP. FIPA. http://www.fipa.org.

FNP02. A. Furfaro, L. Nigro, and F. Pupo. ActorServer: A Java middleware for programming distributed applications over the Internet. In *Proc. of the International Network Conference (INC'2002)*, pages 433–440, University of Plymouth, UK, July 16–18 2002.

FNP06. A. Furfaro, L. Nigro, and F. Pupo. Modular design of real-time systems using hierarchical communicating real-time state machines. *Real-Time Syst.*, 32(1-2):105–123, 2006.

FPV98. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24:342–361, 1998.

Fuj00. R.M. Fujimoto. *Parallel and distributed simulation systems*. John Wiley, 2000.

GCK+02a. R.S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. Dagents: applications and performance of a mobile-agent system. 32, 2002.

GCK+02b. R.S. Gray, G. Cybenko, D. Kotz, R.A. Peterson, and D. Rus. D'Agents: Applications and performance of a mobile-agent system. *Software - Practice & Experience*, 32(6):543–573, 2002.

GKCR00. R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. Mobile agents: Motivations and state-of-the-art systems. Technical Report TR2000-365, Dartmouth College, Hanover, NH, April 2000.

GL87. M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*, pages 677–682, Seattle, WA, 1987.

Gla98. Graham Glass. Objectspace voyager - the agent orb for java. In *Proceedings of the Second International Conference on Worldwide Computing and Its Applications*, WWCA '98, pages 38–55, London, UK, 1998. Springer-Verlag.

GPB+00. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Addison-Wesley, 2000.

GR96. M.P. Georgeff and A.S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.

Har87. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

HCK97. C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile agents: are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, pages 24–47. Springer, Berlin, April 1997.

Hew77a. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

Hew77b. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

HLA. Defense Modeling and Simulation Office. HLA-RTI. http://www.dmso.mil/public/transition/hla.

HN96. D. Harel and A. Naamad. The statemate semantics of statecharts, 1996.

Hoha. F. Hohl. Mobile Agent List. http://mole.informatik.uni-stuttgart.de/mal.html.

Hohb. F. Hohl. RePast online. http://repast.sourceforge.net/repast_3/index.html.

HP98. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1998.

HRHL01. N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents: Summary of an agent infrastructure. In *Proceedings of the Fifth International Conference on Autonomous Agents, Workshop on Infrastructure for Agents, MAS and Scalable MAS*, pages 251–257, 2001.

HS96. A. Haddadi and K. Sundermeyer. Belief-Desire-Intention agent architectures. In *Foundations of Distributed Artificial Intelligence*, pages 169–185. Wiley, New York, NY, USA, 1996.

HW07. Yinfeng Henry and Yu Gabriel Wainer. ecd++: an engine for executing devs models in embedded platforms, 2007.

HZM05. X. Hu, B.P. Zeigler, and S. Mittal. Variable structure in devs component-based modelling and simulation. *Simulation*, 81(2):91–102, 2005.

JA06. M.-W. Jang and G. Agha. Agent framework services to reduce agent communication overhead in large-scale agent-based simulations. *Simulation Modelling Practice and Theory*, 14(6):679–694, 2006.

JAA05. M.-W. Jang, A. Ahmed, and G. Agha. Efficient agent communication in multi-agent systems. In *Proc. of SELMAS 2004*, LNCS 3390, pages 236–253. Springer, 2005.

JAM. JAMES II ONLINE. http://wwwmosi.informatik.uni-rostock.de/mosi/projects/cosa/james-ii?set_language=en.

Jen93. N.R. Jennings. Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

Jen01. N. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

JG00. K.-C Jim and C. L. Giles. Talking helps: Evolving communicating agents for the predator-prey pursuit problem. *Artificial Life*, 6(3):237–254, 2000.

JHZT99. N.F. Johnson, P.M. Hui, D. Zheng, and C.W. Tai. Minority game with arbitrary cutoff. *Physica A*, 269(2–4):493–502, 1999.

JLvR+02. D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, and K. Jacobsen. A TACOMA retrospective. *Software - Practice & Experience*, 32(6):605–619, 2002.

JRT+03. M.-W. Jang, S. Reddy, P. Tosic, L. Chen, and G. Agha. An actor-based simulation for studying uav coordination. In *Proc. of the 15th European Simulation Symposium (ESS 2003)*, pages 593–601, Delft, The Netherlands, October 2003.

JvRS95. D. Johansen, R. van Renesse, and F.B. Schneider. Operating system support for mobile agents. In *In Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, pages 42–45, 1995.

JW98. N.R. Jennings and M.J. Wooldridge. Applications of intelligent agents. In N.R. Jennings and M.J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany, 1998.

KDW00. F. Kuhl, J. Dahmann, and R. Weatherly. *Creating computer simulation systems: An introduction to the High Level Architecture*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.

KFC04. Gary Kratkiewicz, Amelia Fedyk, and Daniel Cerys. Integrating a distributed agent-based simulation into an hla federation. In *In Simulation Interoperabilty Workshop SIW 04*, 2004.

KHH+01. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

KLP03. E. Kofman, M. Lapadula, and E. Pagliero. Powerdevs: A devs-based environment for hybrid system modeling and simulation. Technical report, 2003.

KWD99. F. Kuhl, R. Weatherly, and J. Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

LCZ07. M.Y.H. Low, W. Cai, and S. Zhou. A federated agent-based crowd simulation architecture. In *Proc. of 21st European Conference on Modelling and Simulation (ECMS 2007)*, pages 188–194, 2007.

Lev04. R. Levy. Representing agent and their system: A challenge for current modelling. *Informatica*, 28(1):3–11, April 2004.

LHF05. T. Lu, C. Hsu, and C. Feng. High Level Architecture with mobile data filtering agents. In *Proc. of 2005 Workshop on Compiler*

Techniques for High Performance Computing, pages 83–92, Taiwan, 2005.

LLM+05. Michael Lees, Brian Logan, Rob Minson, Ton Oguara, and Georgios Theodoropoulos. Modelling environments for distributed simulation. In *1st International Workshop on Environments for Multi-Agent Systems (E4MAS), in conjunction with the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS04*, 2005.

LLT07. Michael Lees, Brian Logan, and Georgios Theodoropoulos. Distributed simulation of agent-based systems with hla. *ACM Trans. Model. Comput. Simul.*, 17(3):11, 2007.

LMP04. M. Luck, P. McBurney, and C. Preist. A manifesto for agent technology: Towards next generation computing. *Autonomous Agents and Multi-Agent Systems*, 9(3):203–252, 2004.

LMSW05. M. Luck, P. McBurney, O. Shehory, and S. Willmott. Agent technology roadmap: a roadmap for agent based computing. Technical report, European Coordination Action for Agent-Based Computing (AgentLink III), 2005.

LO98. D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents Aglets.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

Log07. Brian Logan. Evaluating agent architectures using simulation. In *Evaluating Architectures for Intelligence: Papers from the 2007 AAAI Workshop.* AAAI Press, AAAI Press, July 2007.

LT01. B. Logan and G. Theodoropoulos. The distributed simulation of multiagent systems. 89(2):174–185, Feb 2001.

Mae91. P. Maes. The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.

MBB+99. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran1, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In *Mobility: processes, computers, and agents*, pages 628–641. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

MT08. R. Minson and G. Theodoropoulos. Distributing repast agent-based simulation with hla. *Concurrency and Computation: Practice and Experience*, 20:1225–1256, 2008.

NCV06. M.J. North, N.T. Collier, and J.R. Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16:1–25, January 2006.

NNS05. A. Nguyen, T. Nakano, and T. Suda. Learning from nature: Network architecture inspired by biology. *ACM Crossroads*, 11(4):3–7, 2005.

NP01. L. Nigro and F. Pupo. Schedulability analysis of real-time actor systems using Coloured Petri Nets. In G.A. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets – Advances in Petri Nets*, LNCS 2001, pages 493–513. Springer, 2001.

NS76. A. Newell and H.A. Simon. Computer science as empirical enquiry. *Communications of the ACM*, 19(3):113–126, 1976.

Nwa96. H.S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

OWB04. T. Osman, W. Wagealla, and A. Bargiela. An approach to rollback recovery of collaborating mobile agents. *IEEE Transaction on Systems, Man and Cybernetics*, 34(1):48–57, February 2004.

OZ04. A. Omicini and F. Zambonelli. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Sytems*, 9(3):253–283, November 2004.

Pic01. G.P. Picco. Mobile agents: an introduction. *Microprocessors and Microsystems*, 25(2):64–75, February 2001.

Pit. Pitch Kunskapsutveckling AB. prti 1516. http://www.pitch.se/prti1516/default.asp.

PL05. L. Panait and S. Luke. Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

Ple99. S. Pleisch. State of the art of mobile agent computing - security, fault tolerance, and transaction support, 1999.

PR90. M.E. Pollack and M. Ringuette. Introducing the tileworld: experimentally evaluating agent architectures. In *Proc. of National Conference on Artificial Intelligence*, pages 183–189, 1990.

PS97. H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. pages 50–61. Springer Verlag, 1997.

PS09. D. Pawlaszczyk and S. Strassburger. Scalability in distributed simulations of agent-based models. In *Winter Simulation Conference (WSC)*, 2009.

PSS99. H. Praehofer, J. Sametinger, and A. Stritzinger. Keywords discrete event simulation using the javabeans component model, 1999.

PV07. E. Posse and H. Vangheluwe. Kiltera: A simulation language for timed, dynamic structure systems. In *Proc. of 40th Annual Simulation Symposium (ANSS '07)*, pages 293–300, March 2007.

Rao95. A.S. Rao. Decision procedures for propositional linear-time belief-desire-intention logics. In M. Wooldridgen, J. Muller, and M. Tambe, editors, *Proceedings of the Workshop on Agent Theories, Architectures and Languages (ATAL'95)*, pages 1–39, Berlin, Germany, 1995. Springer-Verlag.

RC05. M. Remondino and A. Cappellini. Minority game with communication of statements and memory analysis: a multi-agent based model. *Int. J. of Simulation*, 6(5):42–53, 2005.

rep. Repast projects.

RG91. A.S. Rao and M. Georgeff. Modelling rational agents within a BDI-architecture. In J.F. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 473–484, 1991.

RG95. A.S. Rao and M.P. George. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, San Francisco, USA, 1995.

RK95. S.J. Rosenschein and L.P. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73(1–2):149–173, 1995.

RMDLCMZ09. José L. Risco-Martín, Jesús M. De La Cruz, Saurabh Mittal, and Bernard P. Zeigler. eudevs: Executable uml with devs theory of modeling and simulation. *Simulation*, 85:750–777, November 2009.

SBS00. L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *Proceedings of International Conference on Dependable Systems Networks*, pages 135–143, Washington, DC, USA, June 2000. IEEE Computer Society.

SG96. M. Shaw and D. Garland. *Software architecture: perspective on an emerging discipline*. Prentice-Hall, 1996.

Sim. Simagent.

SPR+08. Mittal S., Zeigler B. P., Martn J. L. R., Sahin F., and Jamshidi. *modeling and Simulation for Systems of Systems Engineering*. John Wiley & Sons, Ltd., 2008.

SR98. B. Selic and J. Rumbaugh. Using UML for modelling complex real-time systems. http://www.rational.com/media/uml/resources/documentation/umlrt.pdf, 1998.

SS03. K.M. Sim and W.H. Sun. Ant colony optimization for routing and load-balancing: Survey and new directions. *IEEE Transactions on Systems, Man and Cybernetic*, 33(5):560–572, September 2003.

SU00. Bernd Schattenberg and Adelinde M. Uhrmacher. Planning agents in james, 2000.

Swa. SWARM. http://www.swarm.org.

Syc98. K.P. Sycara. Multiagent systems. *AI Magazine*, 19(2):79–92, 1998.

Ter. TERRACOTTA. http://www.terracotta.org.

TPA05. G. Tan, A. Persson, and R. Ayani. HLA federate migration. In *Proc. of the 38th Annual Simulation Symposium (ANSS05)*, pages 243–250. IEEE Computer Society, 2005.

TZC+06. G. Theodoropoulos, Yi Zhang, D. Chen, R. Minson, S. J. Turner, Wentong Cai, Yong Xie, and B. Logan. Large scale distributed simulation on the grid. In *Proc. Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops*, volume 2, pages 63–63, 16–19 May 2006.

VA01a. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, December 2001.

VA01b. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. In *Proc. of OOPSLA'01*, 2001.

VCA02. N. L. Vijaykumar, S. V. Carvalho, and V. Abdurahiman. On proposing statecharts to specify performance models. *International Transactions in Operational Research*, 9(3):321–336, 2002.

VCAA06. N. L. Vijaykumar, S. V. Carvalho, V. M. B. Andrade, and V. Abdurahiman. Introducing probabilities in statecharts to specify reactive systems for performance analysis. *Comput. Oper. Res.*, 33(8):2369–2386, 2006.

VMT09. Suryanarayanan V., R. Minson, and G. Theodoropoulos. Synchronised range queries in distributed simulations of multi-agent systems. In *13th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2009)*, Singapore, 2009.

vVJP09. David Šišlák, Přemysl Volf, Michal Jakob, and Michal Pěchouček. Distributed platform for large-scale agent-based simulations. pages 16–32, 2009.

Whi99. J.E. White. Telescript technology: mobile agents. pages 460–493, 1999.

WJ95. M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(1):115–152, 1995.

WJ98. M. Wooldridge and N.R. Jennings. Pitfalls of agent-oriented development. In K.P. Sycara and M. Wooldridge, editors, *Proceedings of the Second International Conference on Autonomous Agents*, pages 385–391, New York, 1998. ACM Press.

Woo97. M. Wooldridge. Agent-based software engineering. In *IEE Proceedings on Software Engineering*, pages 26–37, February 1997.

Woo02. M. Wooldridge. *An introduction to multi-agent systems*. John Wiley & Sons, Ltd., 2002.

Wor07. DEVS World. Devs_world: A platform for developing advanced discrete-event simulation at worldwide scale. In *Internal documen*, 2007.

WPM+05. D. Weyns, H.V.D. Parunak, F. Michel, T. Holvolet, and J. Ferber. Environments for multiagent systems, state-of-the-art and research challenges. In *Environments for Multi-Agent Systems: First Int. Workshop, E4MAS 2004, LNAI 3374*, pages 1–47. Springer-Verlag, 2005.

WPW+97. D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents*, pages 86–97, London, UK, 1997. Springer-Verlag.

WVP+05. Danny Weyns, H. Van, Dyke Parunak, Fabien Michel, Tom Holvoet, and Jacques Ferber. Environments for multiagent systems: State-of-the-art and research challenges. in: Revised papers of the e4mas workshop at aamas04. volume lncs, 2005.

YWCTM09. Liang Y., S.J. Turner W. Cai, G.K. Theodoropoulos, and R. Minson. Interfacing repast with hla using a generic architecture for cots simulation package interoperability. In *Joint SISO/SCS Spring Simulation Interoperability Workshop*, San Diego, CA, 2009.

ZBB+08. A. Zilka, G. Bevin, G. Boner, T. Gautier, O. Letizi, and A. Miller. *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, Berkely, CA, USA, 2008.

ZBC+98. B.P. Zeigler, G. Ball, H. Cho, J.S. Lee, and H.S. Sarjoughian. the devs/hla distributed simulation environment and its support for predictive filtering, 1998.

ZMKK96. B. P. Zeigler, Y. Moon, D. Kim, and J. G.n Kim. Devs-c++: A high performance modeling and simulation environment. In *HICSS*, pages 350–359. IEEE Computer Society, 1996.

ZPK00. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation (second ed.)*. Academic Press, New York, 2000.

ZS03. B.P. Zeigler and H.S. Sarjoughian. Introduction to devs modelling and simulation with java: developing component based simulation models. http://www.acims.arizona.edu, 2003.