A Cache-Aware Environment Integrating Agent-based Simulation with Parallel/Distributed Discrete-event Simulation

by

Yin Xiong

(Under the Direction of Maria Hybinette and Eileen T. Kraemer)

Abstract

We address the challenge of providing a high performance Agent-Based Simulator (ABS). ABS systems offer a convenient API for research in large-scale multi-agent systems, but they suffer from poor runtime performance when compared to alternative multi-agent system simulation methodologies.

ABS provides an effective means for composing computational models that simulate the actions and interactions of autonomous agents (individual or collective entities such as organizations or groups). These simulation systems focus on assessing the complex system as a whole. Research in this field combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming. By improving the performance of these simulation systems we enable research in these areas to move forward more quickly.

ABS has been under development and in use for a few decades, but current ABS programs are usually designed to run on a single machine and they are usually difficult to scale, which greatly hinders its development and application.

Our approach enables ABS systems to run as much as 10X faster than before with no degradation in quality of the result. Our solution is a novel integration of an ABS API with an underlying Parallel/Distributed Discrete Event Simulation (PDES) Executive. PDES provide high performance solutions to a number of applications, including: job allocation and scheduling, load balancing and inter-machine communication.

We designed and implemented an effective, fast and scalable cache-aware environment that allows for a standard agent-based API to leverage a standard PDES simulation kernel. We also invented a novel caching scheme called Computation Block Caching that solves two major problems that traditional caching mechanisms have not solved: caching state variables and returning multiple values for a single key.

Experimental evaluation shows that our cache-aware environment substantially enhances performance and at the same time is easy for application programmers to use.

Index words: agent-based simulation, ABS, parallel/distributed discrete event simulation, PDES, computation block caching, SASSY, TileWorld

A Cache-Aware Environment Integrating Agent-based Simulation with Parallel/Distributed Discrete-event Simulation

by

Yin Xiong

M.A., Wuhan University, China, 1990

M.S., University of Georgia, 2001

M.S., University of Georgia, 2007

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2012

A Cache-Aware Environment Integrating Agent-based Simulation with Parallel/Distributed Discrete-event Simulation

by

Yin Xiong

Acknowledgments

I would like to thank my major professors, Dr. Maria Hybinette and Dr. Eileen Kraemer, for guiding me through my PhD research since 2003, during which time, they not only provided me with academic education and advice, but also showed great sympathy and concern when I encountered severe personal misfortunes such as diabetes, kidney stones and cancers. Without their long-time support, it would be impossible for me to come this far in my PhD program.

I want to thank Dr. John Miller and Dr. Jaxk Reeves who kindly agreed to sit on the advisory committee for my PhD program. Their rich knowledge and professional advice enlarged my horizons in both computer science and statistics.

I also want to thank Glen Matthews and George Vulov, two of my fellow graduate students who worked on and made contributions to SASSY, the simulation kernel I built for my PhD project.

I especially want to express my gratitude toward Dr. Choy and Dr. Hopgood at Nottingham Trent University of UK for providing me with the source code of their DARBS TileWorld that served as the basis of the JTileWorld I built for my PhD project.

Last but not least, I'd like to thank Mr. Ken Powell, the IT manager in out Computer Science Department, who configured 20 virtual Linux machines for me to (remotely) run the experiments for my PhD project. Without his help, I would not be able to carry out my experiments on our departmental facilities.

Table of Contents

Page

Acknowledgments iv

List of Figures vi

Chapter

1 Introduction 1

2 A Review of ABS and PDES 3

2.1 agent-based simulation (ABS) 3

2.2 Parallel/Distributed Discrete Event Simulation(PDES) . 13

2.3 PDES for ABS 24

3 SASSY: the PDES Kernel 33

3.1 Introduction 33

3.2 The Architecture 35

3.3 API to the PDES Kernel 36

3.4 API to the Agent-based Model 38

3.5 API to Monitoring and Steering 41

4 the cache-aware middleware 44

4.1 Introduction 44

4.2 Related Work 46

4.3 Approach 49

5 JTileWorld: the testbed for "situated agents" 58

5.1 Introduction 58

5.2 Related Work 59

5.3 The JTileWorld 62

5.4 JTileWorld and DARBS TileWorld: A Comparison 66

6 Performance Evaluation 71

6.1 Performance Evaluation Using JPHold 72

6.2 Performance Evaluation using JTileWorld 77

7 Conclusions and Future Work 93

Bibliography 95

List of Figures

2.1 Simulation Models: a Taxonomy 14

2.2 Message Passing During Rollback [84] 25

2.3 The Structure of a Logical Process (LP) Actor [10] 27

2.4 Overview of the Architecture of SPADES [70] 28

2.5 Illustration of PDES-MAS Framework[25] 30

3.1 SASSY: The (S)calable (A)gent(s) (S)imulation (Sy)stem 35

3.2 UML View of the PDES Kernel for SASSY 36

3.3 A Snippet of a Configuration File 37

3.4 A Physical Agent Model 38

3.5 An Agent-based Simulation 39

3.6 An LP in SASSY Serves As a Proxy for a Simulated Physical Agent 39

3.7 An Interest Area, an IMLP and Four LPs 40

3.8 Monitoring and Steering Messages Are Routed Either to the Application Level or to the Simulation Control Level 42

3.9 Incoming Steering and Monitoring Messages Kept in a Cache 42

4.1 LP Caching 47

4.2 Computation Block Caching 48

4.3 A Cacheable Function/Method 49

4.4 Designating a Function/Method as Cacheable 49

4.5 Two Cacheable Computation Blocks 50

4.6 Mark the Beginning and Ending of Cacheable Computation Blocks 52

4.7 Cacheable Method Specifications for Computation Block “getTile” 53

4.8 Workflow of Preprocessing 54

5.1 A 10 x 10 TileWorld 59
5.2 TileWorld Running on DARBS [18] 63
5.3 Rules for Agents in DARBS TileWorld [18] 64
5.4 A 16 x 16 JTileWorld Running on SASSY 65
5.5 A 400-cell TileWorld with 16 Agents, 20 Holes, 40 Tiles and 40 Obstacles . . 67
5.6 Average Time per Move on a Single Processor 68
5.7 Average Time per Move on Multiple Processors 69
6.1 Speedup: Hard Caching for JPHold 74
6.2 Speedup: Soft Caching for JPHold 76
6.3 Cache Miss vs. Cache Hit for Computation Block "getTile" 78
6.4 Cache Miss vs. Cache Hit for Computation Block "fillHole" 79
6.5 Total elapsed times for 1-64 reactive and deliberative agents in SIM_AGENT and HLA_AGENT(single federate) 81
6.6 A linear function fitted to the result of the single-federate case in [47]) 82
6.7 Simulation and RTI phase times for an Agent Federate(64 Reactive agents distributed over 1-16 Nodes)[47] 83
6.8 Total elapsed time for an Agent Federate (64 Reactive and Deliberative Agents distributed over 1-16 nodes) 84
6.9 A polynomial function fitted to the result of the multiple-federate case in [47]) 84
6.10 Total elapsed times for 1-1000 deliberative agents in JTileWorld 85
6.11 Total elapsed times for 1-1000 deliberative agents in JTileWorld distributed over 1-16 PEs 86
6.12 Impact of Cache Hit Rate on the Performance of JTileWorld 89
6.13 Speedup for Cache-on vs. Cache-off over 2-8 PEs 90
6.14 Impact of Caching on Rollback 92

Chapter 1

Introduction

A computer simulation is a computer program that attempts to simulate an abstract model of a particular system. Computer simulations have become a useful part of mathematical modeling of many natural systems in physics, astrophysics, chemistry and biology, as well as human systems in economics, psychology, social science, and engineering. Simulations can be used to explore and gain new insights into new technology, and to estimate the performance of systems too complex for analytical solutions [79].

Agent-based simulation (ABS) is a class of computational models for simulating the actions and interactions of autonomous agents (individual or collective entities such as organizations or groups) with a view to assessing their effects on the system as a whole. It combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming. ABS has been under development and in use for a few decades. Such simulations are well-suited for simulating collective behaviors of components.

However, current ABS applications are often designed to run on a single machine and are usually difficult to scale. A solution to these problems can be found in Parallel/Distributed Discrete Event Simulations (PDES) designed with scalability in mind and usually run on multiple computers. PDES excel in advanced technologies in job allocation, load balancing and inter-machine communication.

To address the scalability and performance deficiencies of agent-based applications, we designed and developed an effective, fast and scalable cache-aware environment that allows for a standard agent-based API to leverage a standard PDES simulation kernel. Our contributions include, but are not limited to:

1. the idea of integrating agent-based simulation with parallel/Distributed discrete event simulation via a cache-aware middleware
2. the innovative concept and method of Computation Block Caching
3. the introduction of monitoring and steering into an integrated environment of agent-based simulation and parallel/distributed discrete event simulation.

Experimental evaluation shows that our environment is scalable, substantially enhances performance, and at the same time is easy for application programmers to use.

This dissertation is organized as follows: Chapter 2 provides a literature review of ABS and PDES; Chapter 3 introduces SASSY, the PDES kernel for our environment; Chapter 4 describes our cache-aware middleware in detail; Chapter 5 introduces JTileWorld, the distributed testbed for "situated agents'; Chapter 6 reports the result of performance evaluation; and Chapter 7 summarizes our contributions and proposes future work.

Chapter 2

A Review of ABS and PDES

2.1 agent-based simulation (ABS)

Agent-based simulation (ABS) is a relatively new approach to modeling systems comprised of interacting autonomous agents. Computational advances make possible a growing number of agent-based applications across many fields, ranging from modeling agent behavior in the stock market and supply chains to predicting the spread of epidemics and the threat of bio-warfare and from modeling the growth and decline of ancient civilizations to modeling the complexities of the human immune system [52].

ABS has a large community including subject-matter experts from AI, biology, economics and many other fields. Its literature offers a rich set of techniques in modeling and simulating real world phenomena [7, 51, 56, 75, 81, 84].

2.1.1 General Introduction to ABS

No universal agreement exists on the precise definition of the term "agent" although definitions tend to agree on more points than they disagree. Agent characteristics are difficult to extract from the literature in a consistent and concise manner because they are applied differently within disciplines. Furthermore, the agent-based concept is a mindset more than a technology, one in which a system is described from the perspective of its constituent parts [12]. The concept of an agent is meant to be a tool for analyzing a system, not an absolute classification, in which entities can be defined as agents or non-agents [72]. For example, some modelers consider any type of independent component (i.e., software, model, individual, etc.) to be an agent. Others insist that a component's behavior must be adaptive in order for it to be considered an agent and reserve the term "agent" for components that can, in some sense, learn from their environments and change their behaviors accordingly.

Nonetheless, from a pragmatic modeling standpoint, several features are common to most agents. The following is a summary of the major features of agents from [92, 27, 24, 82, 52, 14]:

1. Autonomy: Agents are autonomous units (i.e., governed without the influence of centralized control), capable of processing information and exchanging this information with other agents in order to make independent decisions. They are free to interact with other agents, at least over a limited range of situations, and this does not (necessarily) affect their autonomy. In this respect, agents are active rather than purely passive.

2. Heterogeneity: Agents permit the development of autonomous individuals. Groups of agents can exist, but they are amalgamations of similar autonomous individuals spawned from the bottom-up.

3. Active: Agents are active because they exert independent influence in a simulation. The following active features can be identified:

    - Pro-active / goal-directed: Agents are often deemed goal-directed, having goals to achieve (not necessarily objectives to maximize) with respect to their behaviors. For example, agents within a geographic space can be developed to find or follow a set of spatial paths to achieve a goal within a certain constraint (e.g., time), when exiting a building during an emergency.

    - Reactive / Perceptive: Agents can be designed to have an awareness, or sense of their surroundings. Agents can also be supplied with prior knowledge, in effect a "mental map" of their environment, thus providing them with an awareness of other entities, obstacles, or required destinations within their environment. Extending the example above, agents could therefore be provided with knowledge of building exit locations.

- Bounded Rationality: Throughout the social sciences, the dominant form of modeling is based upon the rational-choice paradigm. Rational-choice models generally assume that agents are perfectly rational optimizers with unfettered access to information, foresight, and infinite analytical ability [62]. These agents are therefore capable of deductively solving complex mathematical optimization problems in order to maximize their well being, balancing long-run and short-run payoffs in the face of uncertainty. While rationale-choice models can have substantial explanatory power, some of their axiomatic foundations are contradicted by experimental evidence, leading prominent social scientists to question their empirical validity. However, agents can be configured with “bounded” rationality (through their heterogeneity), to circumnavigate the potential limitations of these assumptions (i.e., agents can be provided with fettered access to information at the local level). In effect, the aforementioned “perception” of agents can be constrained. Thus, rather than implementing a model containing agents with optimal solutions that can fully anticipate all future states of which they are a part, agents make inductive, discrete, and adaptive choices that move them towards achieving goals. For instance, an agent may have knowledge of all building exit locations, but agents will be unaware if all exits are accessible (e.g., some may have become blocked through congestion).

- Interactive / Communicative: Agents have the ability to communicate extensively. For example, agents can query other agents and / or the environment within a neighborhood, via neighborhoods of (potentially) varying size, searching for specific attributes, with the ability to disregard an input that does not match a desirable threshold.

- Mobility: The mobility of agents is a particularly useful feature, not least for spatial simulations. Agents can roam the space within a model. Juxtaposed with agents' ability to interact and their intelligence, this permits a vast range of potential uses.

- Adaptation / Learning: Agents can also be designed to be adaptive, which can produce Complex Adaptive Systems [35]. Agents can be designed to alter (limited to a given threshold if required) their state depending on their current state, permitting agents to adapt with a form of memory or learning, but not necessarily in the most efficient way possible. Agents can adapt at the individual level (e.g., learning alters the probability distribution of rules that compete for attention), or the population level (e.g., learning alters the frequency distribution of agents competing for reproduction).

Thousands of agent-based simulations have been developed by scientists in a variety of professions. We have chosen a few to present here because they fulfill the (majority of the) following criteria:

1. maintained and still being developed;
2. widely used and supported by a strong user community;
3. accompanied by a variety of demonstration models. In some instances the models' programming script or source code is available.

2.1.2 Major Software of ABS

Many agent-based simulations have been designed and developed by scientists for use in their professions. These ABS are usually domain-specific, even project-specific, and cannot be used in other domains or for other purposes. But some other ABS applications are designed to be general purpose, for uses in different domains or to solve a variety of problems. Such ABSs are often called "test-bed","kernel" or "server". The following are the major ABS software in this category.

2.1.2.1 Swarm

Inspired by artificial life, Swarm [56] was designed to study biological systems and attempt to infer mechanisms observable in biological phenomena. As a general purpose simulator, Swarm makes no assumptions about the particular sort of model being implemented and imposes no domain specific requirements. In addition to modeling biological systems, Swarm has been used to develop models for anthropological, ecological, economic, geographical, political and computational science purposes.

In the Swarm system the basic unit of simulation is the swarm – a collection of agents executing a schedule of actions. The swarm represents an entire model. It contains the agents as well as the representation of time. Swarms can themselves be agents.

A typical agent is modeled as a set of rules, responses to stimuli. But an agent can also be a swarm – a collection of objects and a schedule of actions. Hierarchical models can be built by nesting multiple swarms. Because swarms can be created and destroyed as the simulation executes, Swarm can be used to model systems in which multiple levels of description dynamically emerge.

Swarm supports special agents that watch the simulation and gather information. These are called "observer agents" and form a swarm. The model swarm runs as a sub-swarm of the observer swarm.

Swarm is a general-purpose simulation engine that has been used in many domain-specific simulations, but Swarm is only good for applications that have a "swarm" structure.

2.1.2.2 RePast

RePast is the leading free and open source large-scale agent-based modeling and simulation library. Users build simulations by incorporating RePast library components into their own programs or by using the visual scripting environments. There are three production versions of RePast. RePast Py is a cross-platform visual model construction system that allows users to build models using a graphical user interface and write agent behaviors using Python scripting. RePast J is a pure Java modeling environment to support the development of large-scale agent models. It includes a variety of features such as a fully concurrent discrete event scheduler, a model visualization environment, integration with geographical information systems for modeling agents on real maps, and adaptive behavioral tools such as neural networks and genetic algorithms. RePast .NET is a pure C# modeling environment that brings all of the features of RePast J to the Microsoft .NET framework[60].

2.1.2.3 SIM_AGENT

SIM_AGENT is an experimental toolkit supporting a mixture of symbolic (e.g., rule-based) and sub-symbolic (e.g., neural) mechanisms. It was developed at the University of Birmingham, UK [75]. It is intended to support exploration of design options for one or more agents interacting in discrete time. It provides an architecture for autonomous agents with human-like capabilities including multiple independent asynchronous sources of motivation and the ability to reflect on which motives to adopt, when to achieve them, how to achieve them, how to interleave plans, and so on.

The SIM_AGENT toolkit can be used both as a sequential, centralized, time driven simulator for multi-agent systems and as an agent implementation language.

SIM_AGENT is implemented in Poplog Pop-11 [3]. It provides a scheduler which “runs” objects in a virtual time frame composed of a succession of time slices. In each time-slice every object in the world is given a chance to do three things: 1) sense its environment, including creating internal representations derived from sensory data; 2)run processes that interpret sensory data and incoming messages and manipulate internal state; and 3) produce actions or messages capable of influencing other objects in the next time slice.

SIM_AGENT enjoyed popularity in its time as quite a few simulation applications were developed with it and researchers continue to work on SIM_AGENT to make it HLA compliant, hence HLA_AGENT [46].

2.1.2.4 JADE

Java Agent DEvelopment Framework (JADE) [8] was developed at the University of Parma, Italy. It is a software framework to develop agent applications in compliance with the Foundation for Intelligent Physical Agents (FIPA) specifications for interoperable intelligent multi-agent systems. The FIPA standard supports common forms of inter-agent conversations through the specification of interaction protocols, which are patterns of messages exchanged by two or more agents.

JADE uses RMI for its inter-machine communication and provides a GUI for remote management, for example, to start and stop the agents. JADE uses Java Behavior abstraction to model the tasks that an agent is able to perform on a thread-per-agent concurrency basis.

Among the ABS listed here, JADE is the only one that endeavored to be FIPA compliant. It is up to the application programmer to decide how to manage the processing on each machine and the communication between machines, which makes it hard for application programmers to use.

2.1.2.5 MACE3J

MACE3J [30] is a Java-based multi-agent simulation, integration, and development testbed developed at the University of Illinois. It is a highly flexible, Java-based agent simulation system. Scaling up is a main design criterion of the system; it has been run with up to 50 processors and 5000 agents. It relies on a Shared System Image system to provide distributed machines with a consistent image of the model.

MACE3J supports scientific approaches to the study of Multi-Agent Systems as well as practical applications. As a scientifically oriented simulation test-bed, it aims at providing:

1. A selectable combination of deterministic (simulation-driven), user-driven, environment-driven, and/or probabilistic control of simulation events;

2. Flexible data gathering and behavior visualization via user-defined and system-defined probes and data channels;
3. Flexible control and steering of simulations through active user involvement in changing simulation parameters at run-time (blurring the distinction between simulation and enactment and facilitating agent transitions to application); and
4. Reusable components for constructing agents, environments, and experiments, coupled with the ability to flexibly import these components from other projects.

2.1.2.6 COUGAAR

Cognitive Agent Architecture (Cougaar) [34] is an 8-year DARPA-funded effort to explore the potential of a distributed multi-agent system for military logistics. Because of its military nature, Cougaar systems are required to be continuously available, and must degrade gracefully if any component is missing, disconnected or damaged. The network over which Cougaar operates may not be wholly reliable; connectivity and bandwidth are limited, and latency is high. Cougaar also assumes that all hardware is inherently unreliable, and any security or survivability mechanism may fail. In this context, the architecture should be self-maintaining and provide robust distributed failure recovery. Cougaar systems should be amenable to security in all its forms: integrity, confidentiality, authentication, and automatic response to attack.

Cougaar agents are designed to be composed from multiple components, such that their own behaviors are emergent from the interactions of those components. Each agent is complex, long-lived and heterogeneous (in both behavior and software), and runs autonomously and asynchronously with respect to its peers.

Cougaar is composed of multiple layered applications including a Component model, Interaction model, Data model, Service discovery, and a Message Transport Service using protocols such as RMI, CORBA, HTTP, and UDP. Cougaar agents communicate via Blackboard with standard publish/ subscribe semantics. A distributed table called "White Page" maps agent names to network addresses while a directory service called "Yellow Page" supports attribute-based queries.

Cougaar provides a suite of services based on Java Servlets. Cougaar components can register Servlet paths within their agents for HTTP-based web presence. The infrastructure handles cross-agent references transparently through HTTP redirects, creating an interwoven distributed user interface through web links.

The US Army is planning to include Cougaar as a central design point in a new logistics decision support system. The US Army is also exploring using Cougaar to build a military maneuver decision support system.

2.1.2.7 MASON

MASON is a discrete event, multi-agent simulation toolkit in Java developed at George Mason University. It is a single-process discrete-event simulation core and visualization toolkit, designed to be used for a wide range of simulations, but with a special emphasis on "swarm" simulations of a very many (up to millions of) agents [51].

The developers of MASON maintained that Java has an undeserved reputation for slowness and that carefully-written Java code can be surprisingly fast.

So far quite a few applications have been developed with MASON, including Network Intrusion and Countermeasures, Cooperative Target Observation in Unmanned Aerial Vehicles, Ant Foraging, Anthrax Propagation in the Human Body, Wetlands: a Model of Memory and Primitive and Social Behavior. [51].

2.1.3 Pros and Cons of ABS

ABS excels in simulating simultaneous operations and interactions of multiple autonomous entities to re-create and predict the appearance of complex phenomena. Two central tenets of ABS are: 1) Simple behavioral rules generate complex behavior and 2) The whole is greater than the sum of the parts. The following is a summary of the major advantages of ABS:

1. Heterogeneity: ABS naturally supports the representation of information-driven processes. This is primarily achieved by agent communication protocols that closely resemble the way that people communicate. Agents therefore offer the promise of facilitating the rapid construction of systems with low development and maintenance costs.

2. Flexibility: ABS also offers high levels of flexibility and robustness in dynamic or unpredictable environments by virtue of their intrinsic autonomy. For example, agents can be provided with objectives and strategies, abstractions not easily supported by classical object models.

3. Knowledge sharing: Agents facilitate knowledge sharing since they require a common metadata model describing what and where the information is that they use, and how the information translates into domain semantics.

4. Learning: Agents allow learning capabilities to be incorporated in a natural way, enabling agent behavior to change with time based on acquired experience. For example, in a decision support environment, patterns of information usage by individual users might be used to influence the prioritization of future suggested actions in particular situations.

5. Distributed control: Agents are rule-based autonomous entities, thus supporting parallel computations on separate machines.

6. Emergent: ABS facilitates simulation of group behavior in highly dynamic situations, thereby allowing the study of "emergent behavior" that is hard to grasp with traditional methods.

But ABS also has its disadvantages:

1. Analytical intractability: Unlike mathematical modeling, agent-based modeling cannot establish theorems, except through multiple runs covering various initial conditions

and parameters. So for some systems, such as economic systems, that need to establish theorems, agent-based simulation may not be proper.

2. Curse of dimensionality: Agent-based simulation is always multi-dimensional, more complex and more difficult to implement than one-dimensional event-based simulations.

3. Wilderness of simulation results: Agents are designed to be autonomous and possess some intelligence. Agent-based simulation enjoys too many degrees of freedom. The results of a simulation program can be different from one run to another due to such freedom.

4. Hard to program: The creation of autonomous and individualized agents is not an easy job. By nature, programmability is the opposite of individuality. Some systems are composed of entities that cannot be said to be autonomous or intelligent in nature. For example, network packets are inanimate and non-autonomous. They are dispatched by computers or routers and do not have to make their own decisions or learn some knowledge on the way to their destination. For the simulation of such systems, agent-based implementation may be overkill.

5. Performance deficiency: From a software engineering and ease of use point of view, ABS simulators are not "high performance" in the same sense that state of the art PDES systems are. In fact, some agent based simulation systems face serious performance limitations. These limitations prevent ABS researchers from investigating systems with thousands or millions of agents [37].

2.2 Parallel/Distributed Discrete Event Simulation(PDES)

A discrete event is something that occurs at an instant in time. For example, pushing an elevator button, starting a motor, stopping a motor, and turning on a light are all discrete events because there is an instant in time at which each occurs. But a discrete event simulation (DES) can simulate both discrete events and continuous events. When a DES simulates continuous events, it “discretizes” the continuous events by only simulating points in time within a continuous duration.

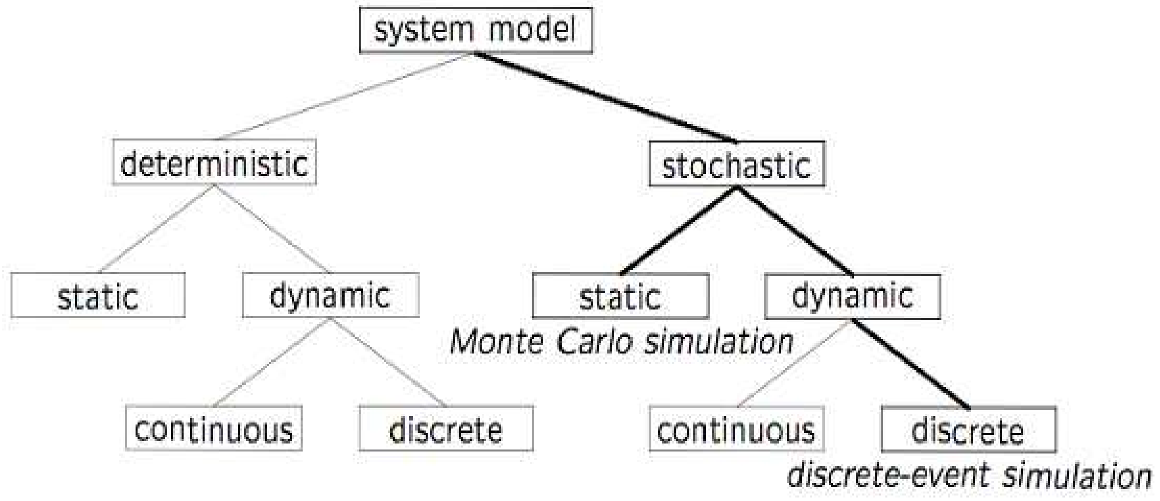


Figure 2.1: Simulation Models: a Taxonomy

DES is a method used to model real world systems that can be decomposed into a set of logically separate processes (LPs) autonomously progressing through time. Each event must take place on a specific process, and must be assigned a logical time (a timestamp). The result of this event can be a message passed to one or more other processes (including, if required, the process on which the event occurred). On arrival at this other process, the content of the message may result in the generation of new events, to be processed at some specified future logical time. The principle restriction placed on DES is that an event cannot affect the outcome of a prior event, i.e., logical time cannot run backwards.

Figure 2.1 is a taxonomy of simulation models provided by [61]. PDES belong to the right branch of the taxonomy tree, namely, discrete event simulation for stochastic events.

2.2.1 General Introduction to PDES

A Parallel/Distributed Discrete Event Simulation (PDES) is a DES that runs on parallel or distributed machines. The idea of parallel/distributed simulation was first proposed by K.M. Chandy and independently by R.E. Bryant. Papers [15] and [13] contain the basic ideas of parallel simulation, the problem of deadlock, and schemes for deadlock resolution, detection and recovery [16]. Alternative schemes proposed by D.R. Jefferson are based on the concepts of Virtual Time [40]. PDES has been a widely researched area in recent years. There are two main objectives in using PDES: first, to increase execution speed and second, to increase memory.

The system modeled by a PDES is viewed as being composed of some number of physical processes that interact at various points in simulated time. The simulation is constructed as a set of logical processes (LPs), one per physical process. All interactions between physical processes are modeled by time stamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes the progress of the process.

In order to assure the correctness of the simulation, PDES must obey the local causality constraint: a discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint if and only if each logical process executes events in non-decreasing time stamp order.

One of two strategies may be used to guarantee that a PDES does not violate the local causality constraint: conservative or optimistic. Conservative approaches strictly avoid the possibility of any causality error ever occurring and rely on some strategy to determine when it is safe to process an event. The optimistic approaches use a detection and recovery approach: whenever causality errors are detected a rollback mechanism is invoked to recover.

PDES can be grouped into three categories: 1) simulation languages; 2) simulation libraries (or: executives, kernels etc.); and 3) simulation applications.

1. Simulation languages: Unlike a PDES library, where the user must explicitly make reference to the library routines provided and often needs to be aware of certain related runtime issues, PDES languages attempt to relieve the user of the need to be concerned with the underlying PDES protocols (Low et al., 1999). The most important PDES languages include:

   - ModSim, an object-oriented simulation language based on Modula-2 [89];
   - YADDES (Yet Another Distributed Discrete Event Simulator), a simulation specification language [65];
   - Maisie, a C-based PDES language developed to support a number of different simulation protocols including both optimistic and conservative [4]
   - Parsec, a language derived from Maisie with several enhancements incorporated and language constructs rewritten [4].
   - Moose (Maisie-based Object-Oriented Simulation Environment), the object-oriented version of Maisie that uses inheritance to support iterative design of efficient simulation models [87];
   - APOSTLE (A Parallel Object-oriented SimulaTion LanguagE), an object-oriented PDES language based on an optimistic protocol [90];

     Though claimed to be easier to use than PDES libraries, PDES simulation languages are in fact difficult to use because each language specifies its own grammar. Learning a new simulation language is not a simple task for scientists whose professions are not computer science.

2. PDES libraries: Many of the PDES library packages are research-oriented and based on optimistic protocols. Compared with conservative protocols, optimistic protocols are much harder to implement, thus attracting more attention from the PDES research community. We reserve the most important PDES libraries for more detailed descriptions in later sections. Here are the ones also worth mentioning:

- Time Warp Operating System(TWOS) is a PDES library implemented in C and funded by the U.S. Army Improvement Program (AMIP) Management Office (AMMO). It is no longer supported since the project was terminated [68].
- The Synchronous Parallel Environment for Emulation and Discrete Event Simulation(SPEEDES) was sponsored by the U.S. Air Force Electronics-Systems Division, Hanscom Air Force Base, MD. It is an object-oriented PDES library implemented in C++ to provide better flow control techniques that facilitate stable execution of optimistic simulations [76].
- CPSim is a C-based commercial PDES library with a conservative simulation kernel that provides for synchronization, scheduling, deadlock prevention and message passing on multicomputer platforms [33].
- ParaSol is an object-oriented PDES library implemented in C++ and developed at Purdue University [54].
- PSK (Parallel Simulation Kernel) is an object-oriented PDES library implemented in C++ and developed at the Royal Institute of Technology, Sweden. It has since been used to study applications specific to mobile telecommunication [71].
- SPaDES (Structured Parallel Discrete Event Simulation) is an object-oriented PDES implemented in C++ and developed at the National University of Singapore [81].
- WARPED is a publicly available Time Warp simulation kernel that defines a standard interface to the application developer and is designed to provide a highly configurable environment for the integration of Time Warp optimizations. It is written in C++, uses the MPI message passing standard, and executes on a variety of parallel and distributed processing platforms [53].

2.2.2 Major PDES Software Projects

The following PDES are selected for more detailed introduction because they are either still being supported, were used extensively, or have made innovative contributions.

2.2.2.1 GTW

Georgia Tech Time Warp (GTW) [22] is a PDES library, implemented using C, which supports an event-oriented world view. It was developed at Georgia Tech. The hardware platform of GTW is assumed to be a cache-coherent, shared-memory machine containing a set of processors, each with a local cache that automatically fetches instructions and data as needed. It focuses on small-granularity simulations that need a simple API that can be efficiently implemented. So the GTW executive only provides a minimal set of basic simulation primitives, while allowing more sophisticated mechanisms to be implemented as library routines.

GTW has been used in many simulation applications. The most prominent advantage is its speed. But it is written in C and is not able to make use of the advantages brought about by object-oriented languages. GTW is platform-specific and not portable. Using optimistic execution that requires state saving, GTW is not suitable for fine-granularity simulations or simulations that have too many state variables.

2.2.2.2 JWarp

JWarp [11] is an optimistic PDES developed at the University of Coimbra, Portugal. It is implemented in Java, making full use of Java's communication-centric, object-oriented, multithreaded serialization mechanism and portability. In JWarp, real world components are modeled as logical processes (LPs). Each LP is assigned to a processor that maintains its own local simulation clock (LVT), a local event list and a set of state variables. Events are modeled as time-stamped messages exchanged between the physical objects of the application (LP).

Mapping real world components to LPs is a common practice among simulation application programmers. For example, a programmer might map a vehicle to an LP, or she might map a road to an LP. But assigning each LP to a processor is rarely seen. Because it maps each LP to a processor, JWarp needs close cooperation from simulation application programmers who should select a manageable unit for "components". For example, if there are millions of ants to simulate, then the simulation application program should not map each ant to an LP. Instead, they should map a swarm of ants to an LP, thus reducing the number of processors needed.

2.2.2.3 IDES

Infrastructure for Distributed Enterprise Simulation (IDES) is a Java-based optimistic distributed simulation engine developed at Dartmouth College and Sandia National Laboratories [57]. Designed for simulating computer systems and communication networks, IDES uses the Breathing Time Buckets (BTB) protocol – a synchronous protocol that does global synchronization periodically revolving around the notion of an "event horizon" similar to Time Warp.

IDES avoided using Java's threads because there is a much greater potential for synchronization errors and race conditions. Instead, it established a two-way socket connection between every pair of processors. A processor creates a thread for each such socket to listen for and deal with incoming messages. A processor also creates a thread to deal with scheduling and executing events. Thus, if there are P processors used in the system, each processor creates P + 1 threads. Potential synchronization problems are then limited to entities that are shared between message handling code and simulation workload code.

As an optimistic discrete-event simulation kernel, IDES possesses the pros common to optimistic PDES. But IDES maps real world components to LPs communicating via sockets, which can induce much more performance overhead than does using threads.

2.2.2.4 Parsimony

The Parsimony Project was developed by Bruno R. Preiss and Ka Wing Carey Wan at the University of Waterloo, Canada [66]. It is a Java-based testbed designed for research in distributed and network-centric computing. It uses RMI as the inter-machine communication mechanism. Parsimony developed 8 simulators: Sequential single-list simulator, Sequential Multi-list simulator, Thread central-scheduling simulator, Thread conservative simulator, Thread optimistic simulator, RMI centralized simulator, RMI Conservative simulator and RMI optimistic simulator. The multiple simulators can be used to explore the advantages and disadvantages of each implementation.

Parsimony has identified some basic requirements for discrete-event simulations:

1. Modeling support: The description of the state and the behavior of an LP is called a model. The simulated system is a network of instantiated models. Therefore, the development language must support the specification of models, the instantiation of models and the concurrent execution of model instances.

2. Dynamic Loading: A simulator must be extensible in the sense that user-defined models can be simulated without requiring the simulator itself to be recompiled.

3. Parallel execution: The modeling paradigm used in PDES leads naturally to an implementation consisting of a network of communicating lPs. Since concurrent programming is difficult to get right, Parsimony does not expect (or require) that the simulation user make use of threads. Rather, the modeling paradigm automatically yields a simulation that can be partitioned for parallel/distributed execution.

4. Marshaling and unmarshaling :In order to build a distributed simulation, it is necessary to be able to distribute model instances over multiple processors and to allow those instances to exchange information. However, both the models and the messages they send are defined by the simulation user. Therefore, the simulator must support

the dynamic distribution of models, which comprise both state (data) and behavior (code), as well as the exchange of arbitrary, user-defined messages. Thus, the development language/ environment must support transparently extensible networking in that it automates the marshaling and unmarshaling of simulation entities (both model instances and messages) [66].

In Parsimony, models are mapped to LPs, which is a common practice among PDES developers. But Parsimony maps events to "runnable objects" each of which runs only once. This mapping seems to be out of the ordinary as it is a common practice to map events to timestamped messages.

2.2.2.5 FATWa

FATWa is an optimistic PDES implemented in Java by Matthew C. Lowry et al. at the University of Adelaide, South Australia [50].

FATWa adopts a modular approach in which different simulation applications, different synchronization mechanisms and different communication mechanisms can be plugged in as modules. It employs objects that act as RMI servers between Java virtual machines (JVM) to make cross-machine calls in which a simulation process is referenced as a parameter. Thus the RMI mechanism automatically serializes the process and de-serializes it at the destination machine.

But FATWa has only been tested with modules migrating processes at random. The issue of migration policies has yet to be investigated. It is not a general purpose simulation kernel. Instead, it focuses on one kind of problem: investigating the interactions between the various classes of optimization that can operate concurrently. Events are usually put into different queues waiting for their scheduled run time. Queuing large numbers of runnable objects could incur much more overhead than queuing timestamped messages.

2.2.2.6 JiST

Java in Simulation Time (JiST) is a discrete event simulation system developed at Cornell University [7]. It uses a new and unifying approach for constructing simulators called virtual-machine-based simulation. JiST executes discrete event simulations efficiently by embedding simulation semantics directly into the Java execution model and transparently performing important simulation optimizations and transformations at the byte-code level. The system provides the standard benefits that the modern Java runtime affords.

The JiST team designed a Rewriter to make use of Java's Byte-Code Engineering Library. After a simulation application is developed and compiled, the Rewriter performs multiple runs over the byte-code, making sure that the application classes are in the desired form, adding some kernel code to the application, and transforming the byte-code to what JiST wants.

JiST is a conservative PDES with no state saving and does not do speculative (optimistic) processing.

2.2.3 Pros and Cons of PDES

Compared with sequential DES, a parallel/distributed DES has the following advantages:

1. Speed: Computation-intensive simulations can run hours or even days. In such cases, PDES can accomplish tasks much faster. This is the main reason that PDES came into being.

2. Scalability: PDES provides better scalability because each computer has limited resources beyond which no more jobs can be handled sufficiently.

3. Reliability: Distributed simulation is also more reliable. If a simulation is running on a single machine, when the machine is down, the simulation is aborted. But with distributed simulation, other parts of the simulation are intact, which may take over the tasks from the crashed machine.

4. Affinity: By “affinity” we mean the resemblance of the system being simulated and the system simulating it. Some systems are parallel or distributed in nature and using PDES to simulate such systems is a natural choice. For example, airport traffic is parallel and distributed in nature because at the same time, many planes are taking off or landing.

PDES also has its disadvantages and problems:

1. Communication overhead: when simulation is distributed over multiple machines, we need some mechanism to pass data and behavior among the machines. This communication can generate substantial overhead especially when the network is not fast or not reliable.

2. Security problems: network security remains a big problem. When a simulation job is distributed over the Internet, the security risk is much higher than if the job were performed on a single machine.

3. Coding and debugging: writing a distributed simulation program is much more difficult than writing a sequential simulation program. It is even more difficult if the distributed simulation is optimistic. The debugging is also hard to do as so many processes are working on the same problem at the same time.

4. Distribution overhead: distributed simulation also incurs some overhead from distributing the simulation job among multiple processors. Also, if information needs to be gathered, the data-collecting process also generates some overhead. So the start-up of a distributed simulation can take quite some time compared to the simulation itself. If the distribution overhead is too much, then the benefit gained from distributed simulation is lost.

5. Causality constraint: Some applications are sequential in nature and exhibit heavy causality. Each nth step could be the cause of the n+1th step. For example, if the

computation of stage 2 completely depends on the result from the computation of stage 1, then it can be challenging to develop a model that adheres to causality constrain because it is not immediately clear on how to develop a "correct" model.

6. Checkpoint overhead: for optimistic PDES, the checkpoint overhead can be substantial if the state size is too big or the event computation is too short or both. Since optimistic PDES needs to save state, a large state will take a substantial time to save, maybe more time than the computation itself. In this case, the performance degrades quickly and the overhead overtakes the benefit.

2.3 PDES for ABS

PDES and ABS are not orthogonal. A PDES can be agent-based, meaning we can use agents to model real world components and even execute the simulation using agents. It can also be event-based, activity-based, process-based, or object-based. An ABS can run on a single-machine DES or a multiple-machine PDES.

2.3.1 General Introduction to PDES for ABS

For more than two decades researchers have been endeavoring to improve the efficiency of discrete event simulation systems (DES). Numerous techniques have emerged such as parallel execution on shared memory multi-processors [22], distributed execution on multiple machines, optimistic execution [39], faster algorithms for Global Virtual Time calculation [28], duplicating (cloning) simulations in progress to aid what-if-scenario analysis [36] and more recently software systems and architectures to improve interoperability between different simulation technologies such as distributed interactive simulation (DIS) and the high-level architecture (HLA) [21] and their extensions.

Researchers from both the PDES and ABS communities have been trying to utilize advanced technologies from each other's field. ABS researchers tried to solve their scalability problem by distributing simulation across multiple machines, while PDES researchers tried to accommodate "deliberative agents" and "situated agents". So far, they have created quite a few PDES for ABS.

2.3.2 Major Software of PDES for ABS

2.3.2.1 JAMES

Java-based Agent Modeling Environment for Simulation (JAMES) [84] was developed at the University of Ulm, Germany. It is one of the few agent-based DES systems that is both distributed and optimistic. The model design in JAMES resembles that of parallel DEVS (Discrete Event System Specification) [94]. Agents have the ability to access and assess their own structure and have beliefs, desires, and intentions.

JAMES associates with each model a simulator and a coordinator, respectively. Thus, the compositional model is like a hierarchy of processors, i.e., simulators and coordinators, controlled by a root coordinator. The simulators form the leaves of the processor tree.

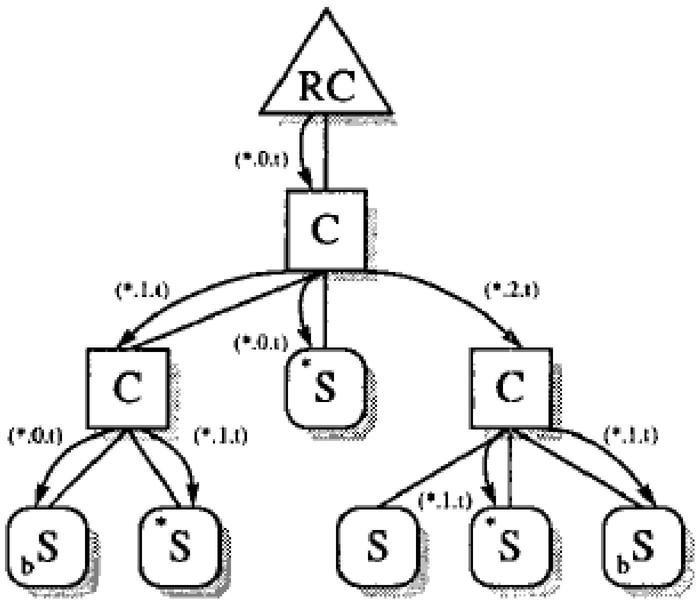


Figure 2.2: Message Passing During Rollback [84]

Figure 2.2 shows the tree-like structure of JAMES with root controller (RC), controllers (C) and Simulators (S).

In JAMES, time is “quasi-continuous” and only those deliberations that happen at exactly the same time are considered “concurrent”. JAMES uses a modified Time Warp scheduling algorithm in their agent-based simulation engine. It’s called “moderate optimism” in that it does not wait until the planner is completed but instead creates a separate external thread and returns a message that indicates that a deliberation process is under way. Thus, the transition function and the overall simulation can proceed. Barrier synchronizations are introduced to prevent the simulation from proceeding too far ahead compared to the external processes still running. To prevent cascading rollbacks over several simulation steps the simulation ensures at each step that it is safe to proceed.

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

JAMES closely follows the Parallel Discrete Event System Specification(PDEVS) formalism, which indicates a hierarchical architecture. An event is not just between two LPs: the sender and receiver, it is about the whole tree-like structure and this is why their optimism can only be “moderate”. It is moderate in the sense it can not speculate too much into the future. It is more like a step-wise optimism: while one thread is doing the current step, JAMES creates a new thread to do the next step. If everything is OK, the next step becomes the current step and the new thread goes ahead to do the next step.

2.3.2.2 TUTW

Temporal Uncertainty Time Warp (TUTW) is an agent-based optimistic PDES developed at the University of Calabria, Italy [10]. It is a control engine designed to make use of temporal uncertainty (TU) in general optimistic simulations, and concentrates on an agent-based implementation that enables distributed simulations over the Internet. A novel concept in TUTW is an event model in which time intervals rather than classical punctual or "precise" timestamps are attached to events. TUTW takes advantage of TU by resolving events so that the number of rollbacks is reduced. The simulator performance can thus be improved without necessarily compromising the accuracy of the results.

TUTW uses ActorFoundry [1] for the modeling and mapping. ActorFoundry is a collection of Java packages and an associated agent-based paradigm for distributed programming directly founded on the Actor model. ActorFoundry facilitates the exploitation of heterogeneous computing environments, e.g., integrating Windows and Solaris platforms. Figure 2.3 displays the anatomy of a Logical Process Actor.

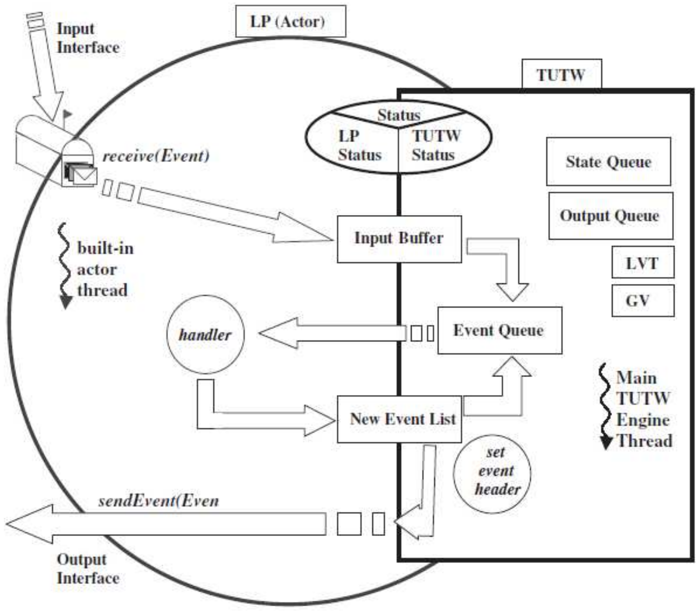


Figure 2.3: The Structure of a Logical Process (LP) Actor [10]

A key feature of TUTW is memory management that avoids dynamic object allocations and associated costly Java garbage collections. To this end, memory pools are maintained separately for each kind of object (events, states, output information, etc.) and consumed objects are saved in these pools rather than released for garbage collection.

For those applications that have no time uncertainty to exploit, the benefit derived from an attempt to exploit time uncertainty would be equal to zero and the overhead brought about by the TUTW algorithm could negatively impact performance.

2.3.2.3 SPADES

The System for Parallel Agent Discrete Event Simulator (SPADES) was developed at Carnegie Mellon University and the Georgia Technology Institute [70]. It is a simulation environment developed for the AI community that focuses on the agent as a fundamental simulation component. The thinking time of an agent is tracked and reflected in the results of the agents' actions by using a "Software-in-the-Loop" mechanism. Figure 2.4 illustrates the architecture of SPADES.

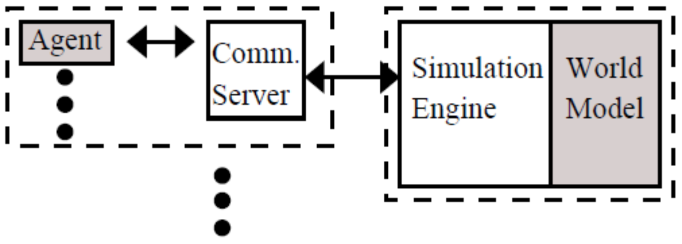


Figure 2.4: Overview of the Architecture of SPADES [70]

Much of the work in creating efficient PDES deals with how to break down a simulation into components such that the communication requirements between the components are low. SPADES takes a different approach. The breakdown of components is fixed (agents and a world model).

SPADES adopts an optimistic synchronization mechanism allowing out-of-order executions as long as they do not violate local causality. Agents' interactions are not necessarily synchronized. Any subset of the agents can have actions take effect at a given time step. This is in contrast to many simulations in the AI community, that require that all agents choose an action simultaneously, with the state of the world model updated once based on all these actions.

SPADES supports agent-based execution, as opposed to agent-based modeling or implementation. Agent-based execution means that the system explicitly models the sensing, thinking, and acting components (and their latencies), which are the core of any agent. SPADES allows arbitrary latencies for each of the time periods, and allows overlapped actions. However, two think cycles are not allowed to overlap, since a typical deployed agent only has a single CPU to use for the thinking step.

SPADES invented the Software-in-the-Loop methodology based on the assumption that thinking time is non-negligible and non-constant. The actual software used in thinking is included as part of the SPADES simulation using Linux perfctr to measure the thinking time. After measuring the CPU time used by the simulated think process and applying a linear scale factor, SPADES schedules the act event at the appropriate delayed simulation time.

SPADES makes no requirements on the agent architecture (except that it supports the sense-think-act cycle) or the language in which agents are written (except that they can write to and from Unix pipes). SPADES provides an environment where agents built with different architectures or languages can inter-operate and interact in the simulated world.

The SPADES system provides reproducible simulation results. Given the same set of initial conditions and the same random seeds, SPADES will produce identical results for every simulation execution.

2.3.2.4 PDES-MAS

Parallel Discrete-event Simulation of Multi-agent Systems (PDES-MAS) was developed by [25] at the University of Birmingham, UK. It is a framework for the distributed simulation of agent-based systems. Each agent in the framework is modeled as an Agent Logical Process (ALP). An ALP has both private and shared state which is accessible to other ALPs. The shared state is modeled as a set of Shared State Variables (SSVs).

AS Figure 2.5, the SSVs are managed by a tree-shaped hierarchical structure of Communication Logical Processes (CLPs), which is dynamically reconfigured to reflect the shared data access patterns in the simulation[25].

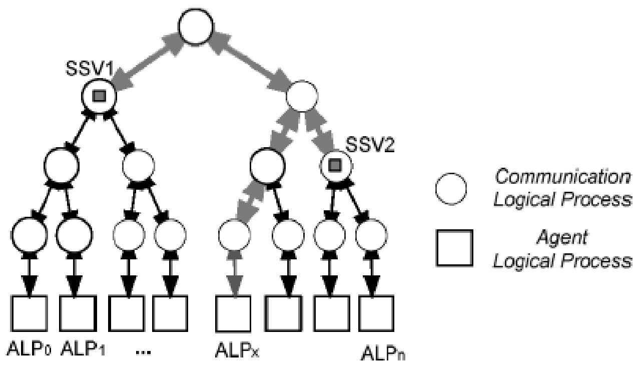


Figure 2.5: Illustration of PDES-MAS Framework[25]

ALPs interact with the shared state and other ALPs through read and write (update value) operations on SSVs which are managed by a tree-shaped hierarchical structure of Communication Logical Processes (CLPs) that are dynamically reconfigured to reflect the shared data access patterns in the simulation. In this process, SSVs are migrated towards the frequently accessing ALPs according to cost measures; thus the scalability of the framework is ensured. A CLP interacts with other LPs via ports. The queries from an ALP are modeled as timestamped messages, for which each CLP acts as a router responsible for forwarding them to the destination CLP(s). The ports are designed to maintain the distribution of the values of SSVs in the value space classified by the types of SSVs.

The application model focuses on the simulation of situated agents, namely, an agent has a position that determines its region of interest: only objects situated in the region can be accessed by the agent. In addition, situated agents are usually able to change their own positions. This behavior was modeled for a two-dimensional environment. An agent moves step-wise towards a pre-selected target along the shortest path, and it randomly chooses a new target on arrival.

PDES-MAS provides Address-based routing and Range-based routing and adopts a meta-simulation approach. Similar to many simulations of P2P systems, the characteristics of the underlying network are abstracted away by only counting hops and messages.

PDES-MAS attempts to incorporate the advantages from both PDES and ABS, or rather, attempts to make PDES work for ABS. The concept and structure of ALP and CLP can be traced back to LP and PE in PDES. It uses an optimistic algorithm more aggressive than JAMES, but their tree-like hierarchical structure for the management of the shared state variables (SSV) hinders the execution of Time Warp mechanisms.

2.3.3 Trade-offs in Creating PDES for ABS

Creating PDES to accommodate the needs of ABS is still a rare undertaking. So far, only a few PDES have done so. From their experience, we identified some trade-offs in creating PDES for ABS:

1. Autonomy vs. Event Scheduling

   Agents are typically defined as objects with autonomy of knowledge, control (decision-making), and interaction. However, LPs in PDES are not "spontaneous" in that they never initialize an event without receiving a message. If we map agents to LPs, the event scheduling mechanism in PDES can be in conflict with the autonomy needed for agents.

2. Autonomy vs. Load Balancing

   In order to get good performance, PDES usually adopt some load balancing mechanisms. One thing PDES commonly do is to create LPs according to user command at the beginning of simulation, then create or delete LPs when necessary. PDES also migrate LPs from machine to machine in order to maintain load balance. In a word, the creation, deletion and migration of LPs are done by the PDES kernel.

But as autonomous entities, agents should be able to create, delete and migrate agents on their own. The trade-off between global load balancing and agent autonomy may depend on the nature of the simulation applications.

Chapter 3

SASSY: the PDES Kernel

SASSY, the Scalable Agents Simulation System, is the PDES kernel of our cache-aware environment.

3.1 Introduction

The PDES paradigm is well suited for simulation applications that consist of multiple computational or processing nodes with packets or messages passing between them. Networking simulators, for instance, represent routers as LPs, and packets as messages/events. Other examples include simulation of air traffic with airports as LPs and aircraft as messages, and road systems as intersections (LPs) and cars (messages). In this paradigm, computation occurs at the fixed LPs and the messages that move between them have no computational capability. Most applications for PDES involve a large number messages in comparison to the number of LPs.

The standard PDES API for simulation developers is not well suited to agent based applications because it does not offer the programming model these researchers expect. For example, multi-agent system (MAS) researchers expect to treat agents as objects that move around in an environment (like messages in DES, but with the ability to compute). In most PDES simulations LPs don't move, they represent geographically static objects such as network routers, airports, sectors in the airspace, intersections. So, in these simulators the objects that perform computing don't move. In contrast, in physical agent simulations the agents move around. According to [70, 5, 32], ABS researchers generally expect their agents to:

1. Use the Sense-Think-Act cycle: Agents sense their environment, consider what to do, then act. This is the predominant computational paradigm for agents; it stands in contrast to the message/event paradigm for PDES.

2. Compute: Agents have computing capability and state; again, in contrast to messages in PDES, which provide no computing function.

3. Proliferate: MAS simulations typically involve hundreds or thousands of agents.

4. Persist: Agents are persistent members of the environment, in contrast to messages that exist only for a short periods.

For these reasons, a number of MAS and multirobot systems researchers have devised their own simulation systems for their research. From a software engineering and ease of use point of view their simulators are well suited to the research tasks they pursue, but these simulators are not high performance in the same sense that state of the art PDES systems are. In fact, some agent-based simulation systems face serious performance limitations. These limits prevent MAS researchers from investigating systems with thousands or millions of agents.

We feel the best solution is to provide middleware between a PDES kernel and agent-based API. This will enable MAS researchers to program using a model that is comfortable for them, while they leverage the high performance of an underlying PDES kernel.

The goal of SASSY is to leverage the efficiency, speed, and parallelism available in PDES systems for use in ABS. SASSY is not the first to make use of PDES for ABS. As previously introduced, others have attempted this task [84, 25, 70]. However, SASSY has several unique aspects that contribute a novel high-performance design. In particular, SASSY uses a "standard" PDES kernel that enables it to easily leverage existing and future performance technologies such as optimistic protocols, distributed execution, and advanced efficient Global

Virtual Time calculations. SASSY also provides a standard ABS API that makes the simulation application developer's job easier: she can more directly map her problem to the simulator without having to know the details of PDES.

3.2 The Architecture

In SASSY, a faster than real time simulation runs on one or more processors, allowing models to advance ahead of the corresponding wall-clock time. Through a web server, Internet users are able to query and steer the simulation. In some cases, as in traffic or multi-robot simulations, users can request specific simulation results for their personal use. At the same time, the researchers who have designed the simulation will be able to revise certain portions of the simulation while it is running.

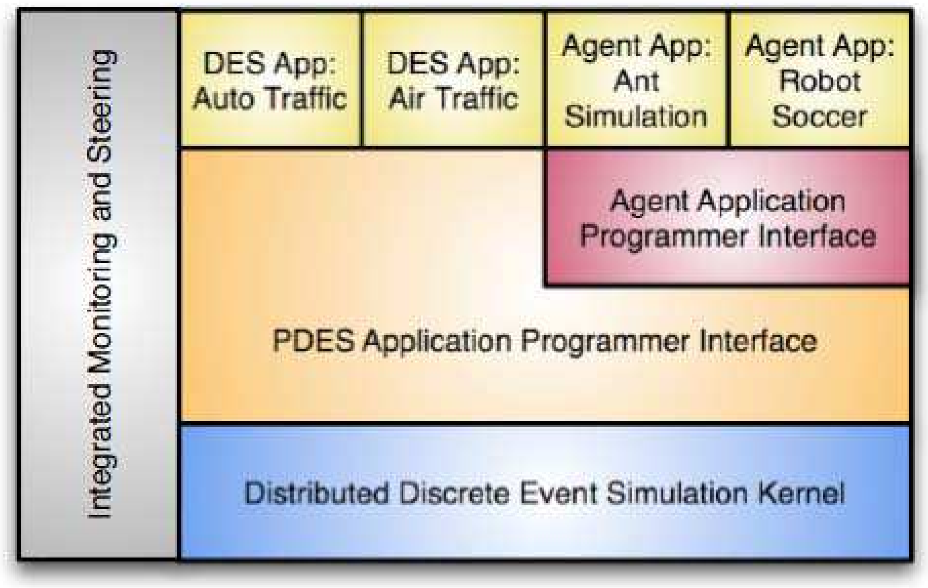

Integrated Monitoring and Steering
DES App: Auto Traffic
DES App: Air Traffic
Agent App: Ant Simulation
Agent App: Robot Soccer
Agent Application Programmer Interface
PDES Application Programmer Interface
Distributed Discrete Event Simulation Kernel

Figure 3.1: SASSY: The (S)calable (A)gent(s) (S)imulation (Sy)stem

The architecture of SASSY is illustrated in Figure 3.1.

3.3 API to the PDES Kernel

The API for the PDES simulation application programmer is simple and easy to use. SASSY is implemented in Java, and therefore benefits from object-oriented system design. The kernel provides an abstract class LP (Logical Process) that implements the features of a generic logical process. Methods requiring application-specific implementation are designated "abstract" and are to be implemented by the application programmer.

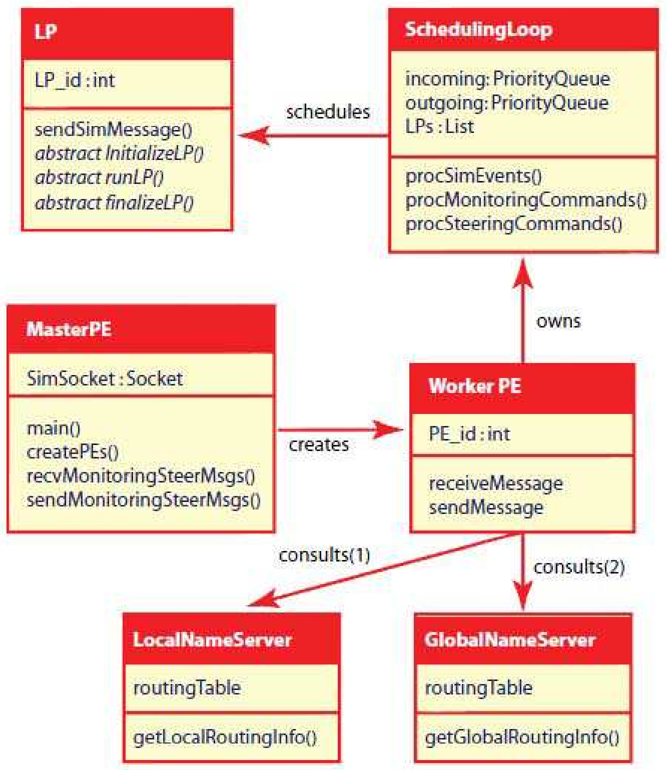
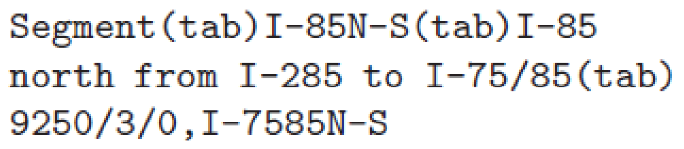


Figure 3.2: UML View of the PDES Kernel for SASSY

To develop a simulation application to run on SASSY, a programmer extends the LP class by implementing the abstract methods: initializeLP, runLP, and finalizeLP (see Figure 3.2). These methods respectively initialize simulation data structures, describe an execution handler that is run when the logical process is scheduled, and call routines that are activated when the logical process leaves the simulation. The LP schedules both remote and local events (including events to itself) via messages.

To run a simulation application on SASSY, two configuration files are needed. One file specifies simulation system configuration information (e.g., number of PEs(processing elements), IP numbers for available machines, etc.). The other file specifies application specifics such as the names of the simulation objects and their corresponding numeric identifiers.



```
Segment(tab)I-85N-S(tab)I-85
north from I-285 to I-75/85(tab)
9250/3/0,I-7585N-S
```

Figure 3.3: A Snippet of a Configuration File

Figure 3.3 contains a single line of a configuration file for a traffic simulation. As the simulation kernel parses the line, it creates an instance of the "Segment" class (which is a subclass of Logical Process), sets its application-level ID to "I-85N-S", sets its description to "I-85 north from I-285 to 1-75/85", and passes the string "250/3/0,I-7585NS" into the Segment's "setConfigData()" method. This method, implemented by the application programmer, parses the string to define this Segment (LP) as 9250 meters long, 3 lanes wide, initially containing 0 cars, and sending all cars leaving this Segment to another Segment (LP) identified as "I-7585N-S". The simulation kernel then assign this newly created LP to one of the PEs.

Monitoring and steering capability permits us to dynamically add (and remove) logical processes (simulation objects) through the monitoring and steering (MS) console.

The SASSY kernel uses a hierarchical name service structure, similar to DNS (Mockapetris, 1987). A global name server (GNS) runs on the MasterPE and contains a mapping of the PE's IDs to their physical addresses. Each WorkerPE runs a local name server (LNS) that maintains LP ID to physical address mapping information for all the LPs running on this PE.

In order to reduce PE to PE communication, each local name server also caches the mapping information concerning LPs on other PEs whenever they acquire such information. Each WorkerPE is responsible for receiving and sending messages. It periodically checks its two queues: the incoming message queue and the outgoing message queue.

For an incoming message, the WorkerPE checks the message type and processes it accordingly. For an outgoing message, the WorkerPE consults the LNS to find the physical address of the destination. If the local name server does not have this information, the WorkerPE consults the GNS to get the physical address and then sends the message to that PE.

3.4 API to the Agent-based Model

In the standard physical agent model, an agent senses its environment, considers what to do, then acts (see Figure 3.6. This is frequently referred to as the "sense-think-act cycle" [70],[84], [25]. Multi-agent simulators are typically configured as shown in Figure 3.5. The code for each agent connects to a process that maintains world state for the simulation.

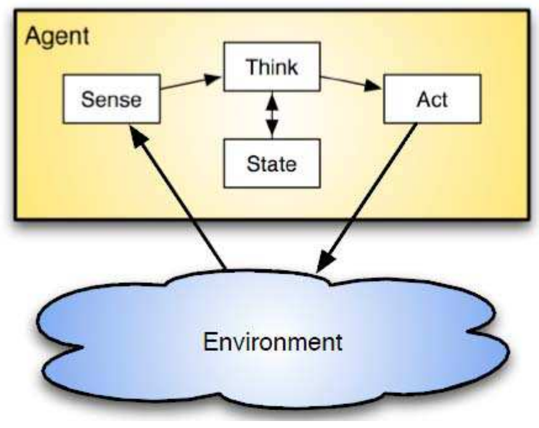


Figure 3.4: A Physical Agent Model

An Application Programmer's Interface (API) allows agents to query the simulator for sensor information and to send actuation commands to the simulator. The simulator updates the world state accordingly. The simulator checks for possible physical interactions that would prohibit a requested action. The simulator also moderates interactions between agents (such as communication).

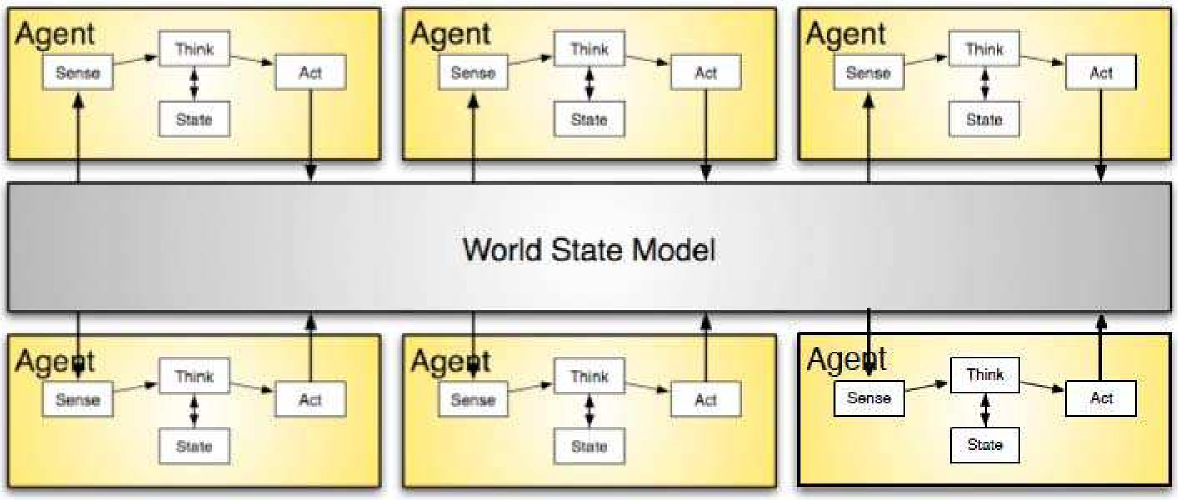


Figure 3.5: An Agent-based Simulation

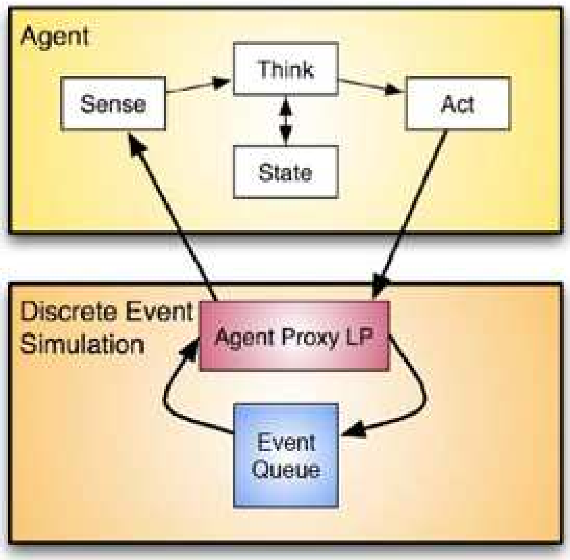


Figure 3.6: An LP in SASSY Serves As a Proxy for a Simulated Physical Agent

Each agent proxy maintains a model of relevant objects in the environment near the corresponding agent for which it serves as a proxy(Figure 3.6). When agents move or act in the world they generate an event that is sent to the other nearby agents so they can track the movements and state of others. The agent proxy LPs keep their state current for the agent they support.

In our approach, there is no central representation of world state. Instead, the world state relevant to each agent is maintained by that agent's proxy LP. As an agent's state changes, it notifies other agents using a state message reflection mechanism. Message reflection is accomplished by a distributed publish/subscribe mechanism implemented by a set of LPs arranged in a grid. These LPs are referred to as Interest Monitoring LPs (IMLPs). Each agent registers interest in (i.e., subscribes to) the activities that occur within specific cells. Agents that move within a specific cell periodically publish their state by sending a message to the relevant IMLP; then the IMLP reflects those messages to other interested agents.

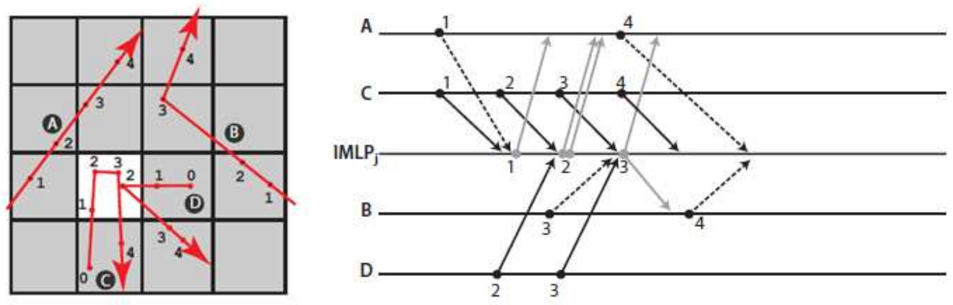


Figure 3.7: An Interest Area, an IMLP and Four LPs

In the picture on the left side of Figure 3.7, four Agents: A, B, C and D, roam about a two-dimensional space. The light colored region is an interest region maintained by $IMLP_j$. Positions of the agents and their directions are denoted by dots and arrows (the numbers refer to instants in simulated time) [37].

The picture on the right side of Figure 3.7 shows the timeline of event messages sent to and from $IMLP_j$. In this timeline, all four agents are roughly synchronized. Events occur as follows: Agent A subscribes to information from $IMLP_j$ at time 1 and unsubscribes at time 4 (all times are given in simulation time). Agent B subscribes at time 3 and unsubscribes at time 4. Agent C enters cell J at time 1 and sends an "enter" message at time 1, then state messages at time 2 and 3. Agent C leaves cell J at time 4 and sends a corresponding “leave” message at that time. $IMLP_j$ receives the state messages from C and reflects them to A and B at the appropriate times. Agent D enters cell J at time 2 and leaves at time 3; it sends appropriate “enter” and “leave” messages at those times. $IMLP_j$ reflects the time 2 state message from D to agent A at time 2. Note that Agent B does not have to be notified of D’s activities because it was not interested in $IMLP_j$ at time 2.

3.5 API to Monitoring and Steering

Different simulation applications have different needs for external runtime input and control (steering) as well as output and display (monitoring). SASSY has a powerful and flexible built-in monitoring and steering (M/S) architecture that can accommodate these varying needs.

When initialized, the simulation creates a socket and listens on a (user-specified) port for connections from a monitoring/steering client program. This M/S client can be implemented in several ways: it can be interactive (as might be needed for run-time adjustment of the simulation parameters either by a human researcher or by a customer steering module such as a machine learning algorithm) or non-interactive (such as feeding in sensor data at periodic intervals for comparison with the simulation’s prediction) and can be implemented in whatever programming language the user prefers, so long as it is capable of sending and receiving through a socket connected to the simulation kernel.

A simple application protocol, modeled after HTTP, is used for the exchange of M/S requests and responses between the M/S client and the simulation. Four types of messages are exchanged: monitoring requests from the client, the corresponding reports from the simulation, steering requests from the client, and the corresponding acknowledgments from the simulation. Although these messages occur as request-response pairs, they are not synchronous, as the client may make a request to be carried out at some future simulation time, in which case no corresponding response will be received from the simulation until the monitoring request is fulfilled.

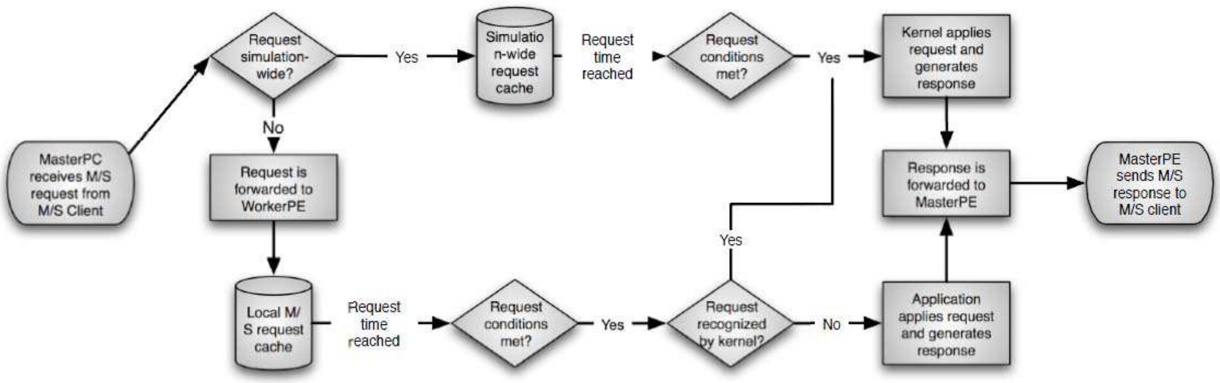

MasterPC receives M/S request from M/S Client
Request simulation-wide?
Yes
No
Simulation-wide request cache
Request time reached
Request conditions met?
Yes
Kernel applies request and generates response
Request is forwarded to WorkerPE
Response is forwarded to MasterPE
MasterPE sends M/S response to M/S client
Yes
Local M/S request cache
Request time reached
Request conditions met?
Yes
Request recognized by kernel?
No
Application applies request and generates response

Figure 3.8: Monitoring and Steering Messages Are Routed Either to the Application Level or to the Simulation Control Level

The monitoring request may initiate a continuing stream of responses rather than a single response. SASSY provides a number of built-in M/S features at the kernel and simulation levels as well as an API for adding additional M/S capability at the application level.

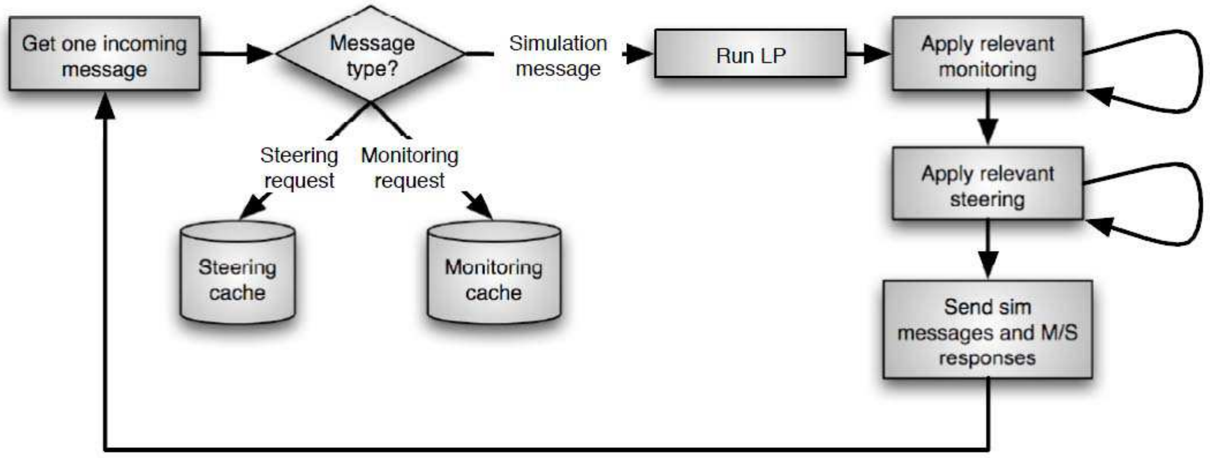


Figure 3.9: Incoming Steering and Monitoring Messages Kept in a Cache

Incoming requests whose types are recognized by the kernel are handled at that level, while unrecognized requests are assumed to be application-specific and are forwarded to the application for handling (See Figure 3.8 and Figure 3.9 for details). Handling at this point consists of scheduling the monitoring and steering requests in the appropriate request cache. Monitoring and steering actions occur only at their scheduled times and when their conditions, if any, are met. This approach permits monitoring and steering actions to be performed on a one-time, ongoing, periodic or conditional basis.

The M/S architecture is linked with SASSY from the lowest levels of the simulation kernel to the application level, allowing monitoring and steering flexibility. It can monitor and steer at various levels: the application level variables, simulation level logical processes (e.g., scheduling semantics) and the simulation itself (e.g., stop, pause, or replicate the whole simulation). In other words in addition to fine-grained application-level monitoring and steering, e.g., observing and adjusting individual application variables, it is also possible to monitor and steer the simulation itself at a coarse-grained level, even to the point of controlling and observing powerful kernel features such as load-balancing, cloning, and merging of simulations.

Chapter 4

the cache-aware middleware

In order to enhance the performance of SASSY, we designed cache-aware middleware that provides transparent, flexible and adaptive caching approach to reduce redundant computations.

4.1 Introduction

Caching the results of expensive and redundant computations or database retrievals improves application scalability and execution time. The idea of caching is not new but has been around since the 1960s when it was first introduced to improve the performance of the Model 85, part of the System/360 IBM product line. Typical PDES systems re-compute events in time stamp order, without exploiting a computational result cache even if identical events may have been processed earlier. It is thought that for most such simulations events are fine grained (light weight) computations and that the cost-savings of reusing the results of such computations would not sufficiently offset the overhead of caching to provide an improvement in performance. However, ABS events are usually coarser-grained than the events assumed by traditional PDES systems.

PDES events typically require less than a millisecond [77, 22], while ABS events typically run for tens of milliseconds or longer. This is because ABS involves deliberative agents as well as reactive agents. While reactive agents are similar to reflex actions in that they simply retrieve pre-set behaviors without maintaining any internal state, deliberative agents behave more like they are thinking, by searching through a space of behaviors, maintaining internal state, and predicting the effects of actions. Agent-based simulations of robots (e.g., TeamBots[5] and Player/Stage [32]) often assume a time step rate of 33 msec as this corresponds to the frequency at which a video camera delivers images. Further, all of the intervening time is typically used to process the information and compute a movement command. However, these agent-based simulations do not scale well.

Agents in an ABS normally rely on a sense-think-act cycle. Agents sense the environment, consider what to do, and then act.

TileWorld, a test bed to evaluate the reasoning of agents, requires substantial time to deliberate [63]. The thinking step, independent of a particular planning algorithm, as observed by the agent-based simulation community, ranges from a complex step requiring lengthy computation (e.g., 1 second [84] or 10 ms to 1000 ms [6]) to a reactive step with negligible "thinking time". Accordingly, the performance of an agent-based simulation can be improved significantly by speeding up the lengthy thinking process. We exploit variable thinking time and use adaptive caching in which we cache the input parameters and the results of lengthy thinking in order to avoid re-computation, but avoid caching computations where the relevant time is trivial, such as with reactive agents that do not think, where caching may not be worth the cost.

An agent's thinking process may involve several input parameters and possibly depend on a large state space, and the probability of encountering exactly the same set of parameters and state variables can be low. Thus, caching the ultimate result of the whole thinking process may not be beneficial as the cache hit rate can be minimal. Here, our approach of block caching enables breaking the thinking process into smaller units that may be more amenable to a caching mechanism and less (as a whole) dependent on the state space.

We designed a novel caching scheme called "Computation Block Caching'. Our caching is flexible and transparent to the application developer, as it requires no additional coding or recoding. By using a software cache Preprocessor, caching code is integrated and compiled automatically. Our motivation is to make caching transparent to the user while improving scalability and performance.

4.2 Related Work

Caching is used in different applications and is integrated at different levels into the architecture including software, language systems and hardware. Function caching or memoization is a technique suggested by the programming language research community to improve the performance of functions by avoiding redundant computations. Here, function inputs and corresponding results are cached in anticipation of later reuse [67].

Function caching is used for incremental computations, dynamic programming and in many other situations. Incremental computations allow for slight variations in function input. It makes use of previous results and adjusts them to generate new output. Using function caching to obtain efficient incremental evaluation is discussed in [67]. Deriving incremental programs and caching intermediate results provides a framework for program improvement [48]. Memoization is available today as part of the Java programming language.

Walsh and Sirer proposed simulation staging, a form of function caching, as a way to improve the performance of a sequential discrete event simulation in applications with a substantial number of redundant computations [88]. Their approach provides significant speedup (up to 40x in a network application), but requires extensive structural revision of code at the user application level.

Contrary to our approach, function caching techniques do not consider the cost of consulting the cache and are not adaptive. Observe that if the cost of checking the cache exceeds the cost of just doing the computation, caching will degrade performance. Function caching also relies on an assumption of no side effects (e.g., by variables in the state space) and that the function produces only one output.

The PDES community has proposed different techniques of reusing computations. In cloning [36], simulations cloned at decision points share the same execution path before the decision point and thus only perform those computations once; after the decision point simulations can further share computations as long as the corresponding computations across the different simulations are not yet influenced by the decision point.

Updateable simulation, proposed by [26], updates the results of a prior simulation run, called the base-line simulation, rather than re-executing a simulation from scratch. A drawback of this approach is that one must manage the entire state-space of the baseline simulation. Both cloning and updateable simulation are appropriate for multiple similar simulation runs.

Lazy evaluation, another related approach used in optimistic simulators to improve the performance of rollbacks, caches the original event in anticipation of it being re-used after a rollback and thus avoids re-computation[89]. However, lazy evaluation is only beneficial for events on the same execution path.

Our group developed LP caching [20] for parallel and distributed simulators. Both LP caching and Computtion Block Caching are independent of the simulation engine (i.e., it may be used with both conservative and optimistic simulation kernel). The caching middleware is situated between the PDES kernel and the simulation application. When the kernel delivers an event to the application, the caching software intercepts it. In the case of a cache hit, the retrieved result is passed back to the kernel without the need to consult the application code. This scheme is adaptive in the sense that it avoids consulting the cache when the computation is negligible.

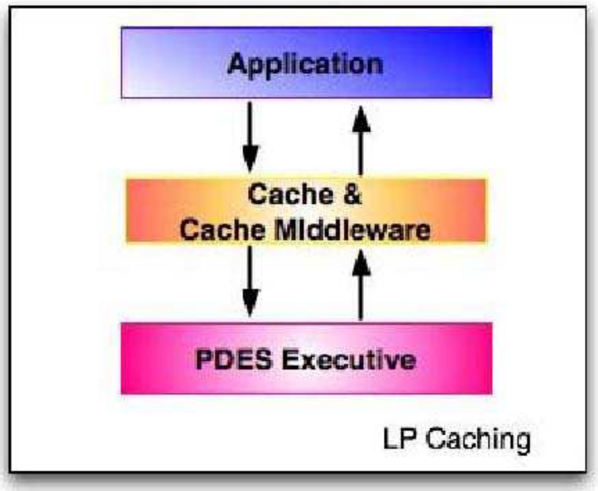


Figure 4.1: LP Caching

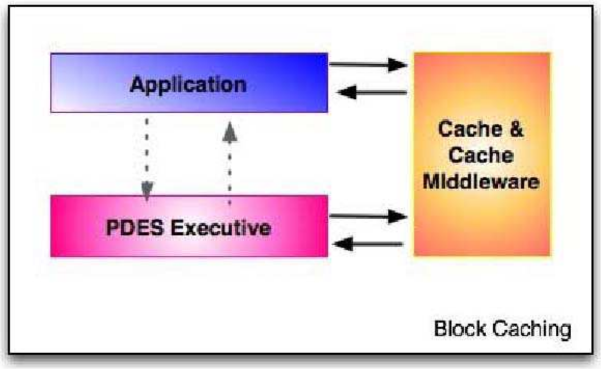


Figure 4.2: Computation Block Caching

A significant difference between LP caching and block caching is LP caching (Figure 4.1) exploits the PDES paradigm of logical processes (LPs) and messages while Computation Block Caching is paradigm independent and can be plugged into a variety of applications and application levels. Figure 4.2 illustrates how this cache-aware middleware is integrated with a PDES simulation.

Our goal of transparency is inspired by JiST [7], which infuses sequential discrete simulation semantics directly into the Java Virtual Machine (JVM) to provide a transparent user programmer interface. In JiST a rewriter reprocesses or rewrites simulation application class code in order to incorporate embedded simulation time operations. The rewriter is a dynamic class loader. It intercepts all class load requests and subsequently verifies and modifies the requested classes. The program transformation occurs once, at load time, and does not rewrite the code during execution. Although JiST does not provide caching functionality we hope in future work to explore embedding our caching middle-ware into the JVM to improve the interface and further transparency.

We designed and developed a cache-aware middleware that provides computation-block caching, a transparent, flexible and adaptive approach to reduce redundant computations. It is transparent in the sense that no recoding is required on the part of application programmers. It is flexible since it can decompose large computations into smaller and potentially re-order to improve performance. It is adaptive in the sense that the caching mechanism is turned on when statistics shows that the benefit of caching exceeds that of computation by a pre-specified factor. In the next sections we will discuss the approach and implementation.

4.3 Approach

We define a "computation block" to be a chunk of code that may be a function/method or a number of lines of code with or without invocations of functions/methods. Computation Block Caching is not as rigid as traditional function caching. It allows state variables to be involved in caching and the result it returns is not limited to returning the value of a single variable.

A computation block can be a function/method. For example, the code in Figure 4.3 is a Java method we implemented for our JPhold benchmark program.

```
int fibonacci  (int  k)  {
    // Base Case:
    if (k <= 2) {
        return 1;
    }
    // Recursive Case:
    else {
        return fibonacci(k-1)  + fibonacci(k-2);
    }
```

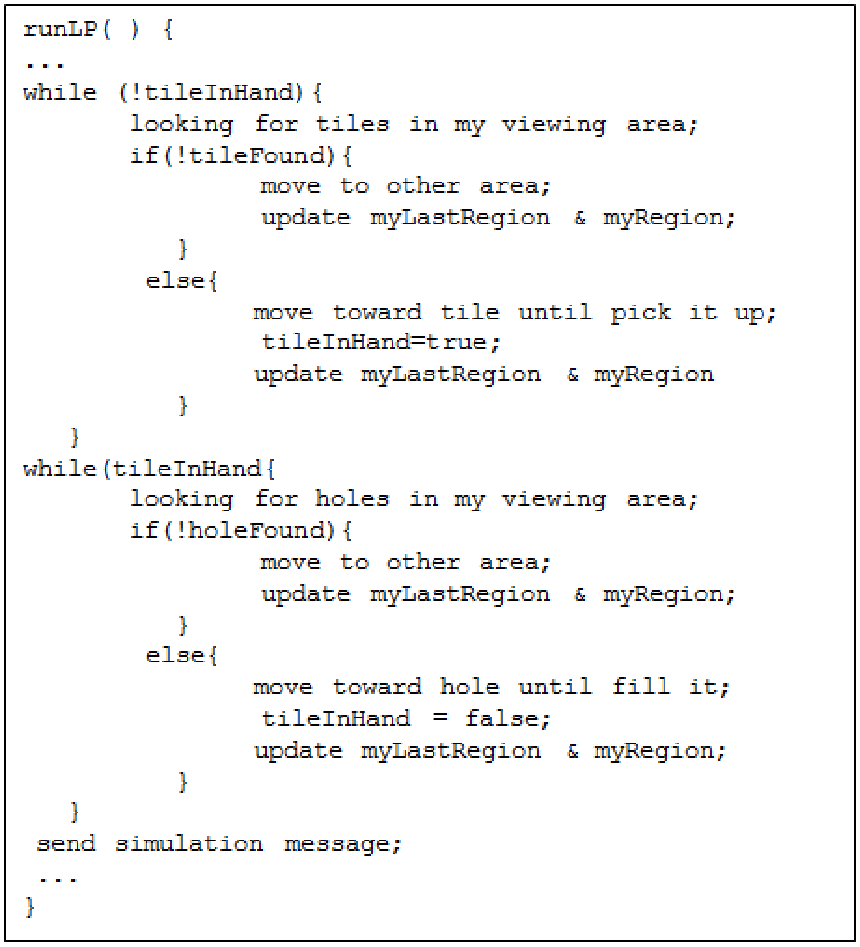
Figure 4.3: A Cacheable Function/Method

To make a function/method cacheable, we specify the method name and other required variables in XML manner as Figure 4.4 illustrates:

```
begin: fibonacci
       packageName:  app
       className:  JPhold
       methodName:  fibonacci
       return:  int result
       parameters:  int  k
       stateVariables:  NA
       cachingFlag:  on
end: fibonacci
```

Figure 4.4: Designating a Function/Method as Cacheable

But our Computation Block Caching can do more than function caching. Consider the computation blocks listed in Figure 4.5:



```
runLP( ) {
...
while (!tileInHand){
        looking for tiles in my viewing area;
        if(!tileFound){
                  move to other area;
                  update myLastRegion & myRegion;
           }
         else{
                 move toward tile until pick it up;
                  tileInHand=true;
                 update myLastRegion & myRegion
           }
   }
while(tileInHand{
        looking for holes in my viewing area;
        if(!holeFound){
                  move to other area;
                  update myLastRegion & myRegion;
           }
         else{
                 move toward hole until fill it;
                  tileInHand = false;
                 update myLastRegion & myRegion;
           }
   }
 send simulation message;
 ...
}
```

Figure 4.5: Two Cacheable Computation Blocks

For traditional function caching, the two "while" loops in Figure 4.5 are not easily cacheable because they violate the basic rules for function caching, namely, they are not a function, but involve multiple functions and state variables. However, the simulation application may have every reason to want to cache these two blocks of code.

One way for traditional function caching to solve the problem is to cache the functions separately, but the amount of recoding will be substantial as each function will need some recoding in order to make it cacheable. Furthermore the functions may write or read from variables that are not passed in as parameters (e.g., "tileInHand" and "tileFound"). The state variables that affect the functions need to be denoted and their updated values need to be copied back, which requires more recoding.

Block caching relieves the application programmer of the tedious task of recoding by utilizing the Preprocessor that automates the recoding process by generating a new version of the code on-the-fly.

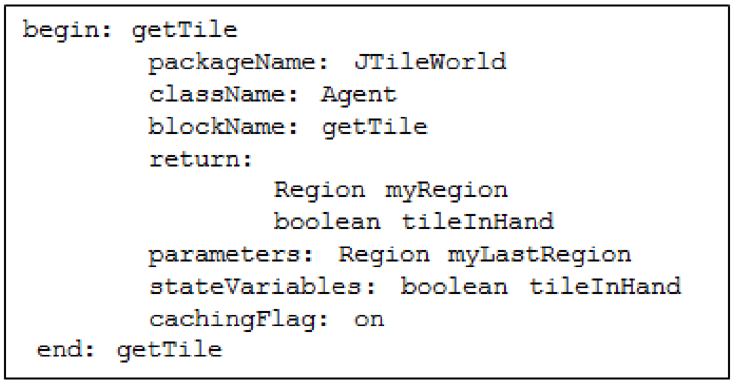
To make a computation block “cacheable”, we first mark the beginning and ending of the block in the application file and then specify in XML manner the block name and other variables required for a cacheable computation block.

```
runLP( ) {
...
//beginCacheableBlock:  getTile
while (!tileInHand){
       looking for tiles in my viewing area;
       if(!tileFound){
               move to other area;
               update myLastRegion & myRegion;
          }
        else{
               move toward tile until pick it up;
               tileInHand=true;
               update myLastRegion & myRegion
          }
   }
//endCacheableBlock:  getTile

//beginCacheableBlock:  fillHole
while(tileInHand{
       looking for holes in my viewing area;
       if(!holeFound){
               move to other area;
               update myLastRegion & myRegion;
          }
        else{
               move toward hole until fill it;
               tileInHand = false;
               update myLastRegion & myRegion;
          }
   }
//endCacheableBlock:  fillHole

 send simulation message;
 ...
}
```

Figure 4.6: Mark the Beginning and Ending of Cacheable Computation Blocks

As Figure 4.6 shows, marking the beginning and ending of a cacheable computation block is simple: using the key words “beginCacheableBlock” and “endCacheableBlock” followed by the name you give to the block. This is not recoding as it does not require re-compilation.

Next, in a text file we specify the cacheable computation blocks in XML style. The following is the full-length specification for the “getTile” block we just marked in the code of the runLP() method. Similar to the specification of a cacheable function/method, the specification for a chacheable computation block needs the package name, class name, the names and data types of the return variable, the parameters and the state variables. The only difference is that "blockName" is used instead of "methodName".



```
begin: getTile
        packageName:  JTileWorld
        className:  Agent
        blockName:  getTile
        return:
                Region  myRegion
                boolean  tileInHand
        parameters:  Region  myLastRegion
        stateVariables:  boolean  tileInHand
        cachingFlag:  on
 end:  getTile
```

Figure 4.7: Cacheable Method Specifications for Computation Block "getTile"

The specification illustrated in Figure 4.7 tells the Preprocessor what should do with the computation block named "getTile", which will be elaborated on in the next section. The specification for "fillHole" is similar and not displayed here.

4.3.1 The Caching Middleware

The cache middleware is composed of two modules: a Preprocessor that reads a configuration file and generates code before a simulation run and a cache manager that manages caching. Figure 4.8 depicts the interactions between the caching modules and a pre-existing PDES simulation kernel and its simulation application.

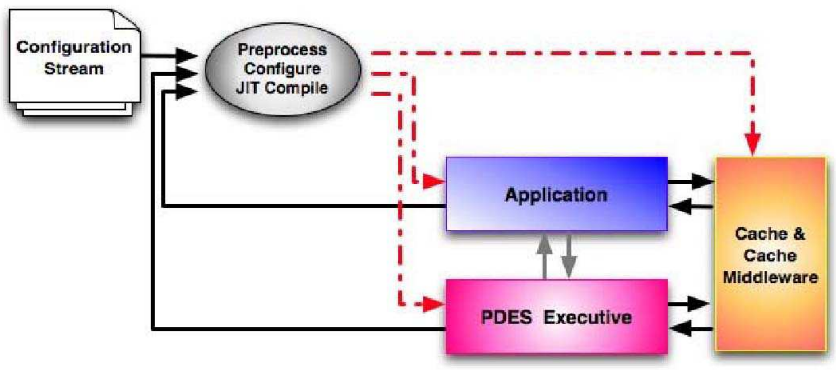


Figure 4.8: Workflow of Preprocessing

4.3.2 The Preprocessor

Existing caching schemes are not suitable for our purposes because they usually require substantial recoding in order to use the caching facilities. By “recoding”, we mean manually modifying the code of the cacheable functions, such as adding, deleting or rewriting lines of code. Therefore, such caching schemes involve “hard coding” which can be error-prone and time consuming. For cacheable functions, the recoding is usually on a function-by-function basis, i.e., for each cacheable function, the application programmer needs to do some recoding in order to make that function cacheable. For example, in [20], a cacheable function needs at least 4 lines of recoding. For LP caching, a 4-line recoding may not be too much as it caches only one function per LP. But for Computation Block Caching, LP events can be decomposed into multiple cacheable computation blocks (note that decomposing a function may also make chunks of code less dependent on state variables and less dependent on one another if reordering is advantageous). For a multi-computation-block program, if each computation block needs 4 lines of recoding to make it cacheable, the total amount of recoding may render the task intimidating and time consuming. Furthermore, for many computation blocks, 4 line of recoding is far from enough.

The Preprocessor in block caching completely relieves the application programmer of recoding in order to make a computation block cacheable. As Figure 4.8 shows, the Preprocessor reads the configuration file and generates a new version for each of the involved application files, inserting caching-specific code into the places specified by the configuration file.

Take the computation block named “getTile” as an example, in the preprocessing phase, the Preprocessor rewrites the “Agent.java” file according to the specification illustrated in Figure 4.7 by inserting a small chunk of code immediately after the line with the key words “beginCacheableBlock: getTile”. What the chunk of code does is as follows:

1. consult the cache
2. if it is a cache hit, return the result fetched from the cache
3. otherwise go on with the original code

The Preprocessor also inserts some code immediately before the line with the kewy words “endCacheableBlock: getTile”. The code is to get the most updated values of the variables designated by the application programmer in the specification file (see Figure 4.7, generate an “input-output” pair and put into the cache for future use.

A simulation application can designate multiple computation blocks as “cacheable”. A cacheable computation block need not always be cached all the time. The user can specify which computation blocks are to be cached for a certain simulation run by adjusting the “cachingFlag” in the specification file.

After the Preprocessor completes rewriting, the Global Name Server (GNS) compiles the modified code and then runs the simulation. There is no need to invoke the Preprocessor for each simulation run. It is invoked only when the specification for the cacheable computations is modified.

The time for preprocessing is decided by a few parameters: the number of cacheable computation blocks, the number of class files, and the length of class files. The Preprocessor scans the configuration file to find which application Java files are involved in caching, then reads the files one by one and inserts caching-specific code at the right places.

4.3.3 The Cache Manager

The cache manager manages caching and determines whether to consult the cache or not. Figure 4.8 depicts the interactions between the caching modules and a pre-existing PDES simulation kernel and its simulation application. The Preprocessor first reads a configuration file or stream (a stream if it generates code while the simulation is running) then "recompiles" the affected objects (dashed arrows in Figure 4.8 denote the flow of output of code to the affected modules).

The regenerated code enables the cache middleware to intercept and monitor cacheable function calls (or blocks) in both the simulation kernel and the simulation application.

To provide user control over whether functions or blocks are cached, a caching flag can be marked as "on" or "off" on a per block basis. A block's flag may be changed at any time, before the application runs or while it is running. The state of the flag (on or off) is set in the configuration stream.

The user can also use the global cache flag to determine the type of cache for each simulation run. The global cache flag can be set in the command line, with 0 = no caching; 1 = hard caching and 2 = soft caching.

The cache is implemented as a Java HashTable and is indexed by the combination of package name, class name, computation block name, passed-in parameters and the state variables involved in the computation. The result of the computation is stored with the index as a key-value pair in the hash table. Our caching middleware supports both conservative and optimistic simulation kernels. It can also be used with both ABS simulation and non-ABS simulations. No changes are required for the simulation kernel or the simulation application.

4.3.4 The Statistics Manager

A feature of our method is that it allows both "hard caching" and "soft caching" options. By "hard caching" we mean that the "cache flag" is set "on" or "off" before a simulation begins to run and remains so throughout the simulation. With “soft caching”, the “cache flag” is set and reset on-the-fly during the simulation.

The Statistics Manager help in both hard caching and soft caching. The Statistics manager is composed of two sub-managers. One sub-manager computes the average caching overhead and the cost for each cacheable computation block on the target computer system. A default program is provided for measuring the caching overhead on the target system. An interactive user interface is provided so the user can specify the range and distribution of each parameter and each state variable for their cacheable computation blocks. With this information and the specification of the cacheable computation blocks, the Statistics Manager creates a stub program for each of the computation blocks, generates parameters and state variables according to the user-specified ranges and distributions, executes each computation block 100 times and reports the average time as the cost for each computation block. The user can compare the computation cost with the caching overhead to decide whether the “caching” flag should be turned on.

The other sub-manager gathers information about the parameters, state variables and the cost of the computation as the simulation is running. It then decides whether the caching flag should be turned on or off for a certain cacheable computation. If the benefit of caching surpasses a certain threshold specified by the user beforehand, or generated on-the-fly, caching will be turned on, otherwise, it will be turned off.

Chapter 5

JTileWorld: the testbed for "situated agents"

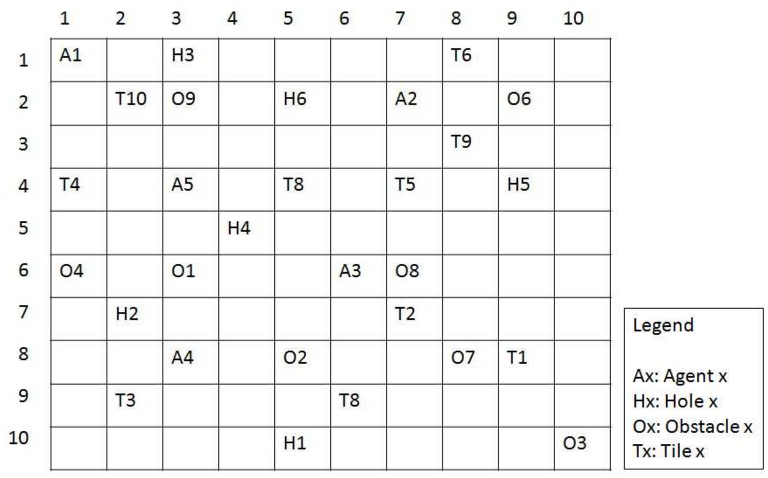
Benchmark programs such as PHold are usually "one-dimensional" in the sense that they only deal with the time dimension because it does not matter where the LPs "situate". But many agent-based simulations contain "situated agents" such as people, cars, swarms of ants, and tile workers etc., moving around in a two-dimensional or three-dimensional world. Even the so-called "inanimate objects", such as holes or obstacles, have their positions. The positions of agents and inanimate objects play important roles in simulations involving "situated agents".

In order to make our cache-aware environment truly ABS-friendly, we built JTileWorld, a Java version of TileWorld, for two purposes:

1. to demonstrate that our cache-aware environment can accommodate "situated agents";
2. to test our Computation Block Caching on a real world ABS in addition to PDES benchmark programs.

5.1 Introduction

The TileWorld is a two-dimensional grid. Figure 5.1 represents a 10 x 10 TileWorld in which agents, tiles, holes, and obstacles exist. The objective of the agents is to score as many points as possible. They score points by moving around the TileWorld to find and pick up tiles, which they put into holes. The agents have a limited view of the TileWorld. The viewing radius is a variable that can be set beforehand or during the run. Each cell can only be occupied by one entity (e.g., an agent, a hole, a tile or an obstacle) at a time. Agents cannot move to a cell with an obstacle in it. The holes, obstacles, and tiles in the TileWorld can change dynamically, i.e., they can appear and disappear in different locations in the TileWorld. The rate of change is set by a variable, and this can be used to reflect the dynamically changing real-world environment.



	1	2	3	4	5	6	7	8	9	10
1	A1		H3					T6		
2		T10	O9		H6		A2		O6	
3								T9		
4	T4		A5		T8		T5		H5	
5				H4						
6	O4		O1			A3	O8			
7		H2					T2			
8			A4		O2			O7	T1	
9		T3				T8				
10					H1					O3

Legend

Ax: Agent x
Hx: Hole x
Ox: Obstacle x
Tx: Tile x

Figure 5.1: A 10 x 10 TileWorld

In TileWorld both the agent and the environment are highly parameterized, enabling one to control certain characteristics of each. Users can experimentally investigate the behavior of various meta-level reasoning strategies by tuning the parameters of the agent, and can assess the success of alternative strategies in different environments by tuning the environmental parameters.

5.2 Related Work

TileWorld has been widely used to test the behavior and interaction of multiple agents in a dynamic environment [43, 64, 38]. Using TileWorld as a testbed offers many advantages, including but not limited to the following:

Table 5.1: A Summary of Five TileWorlds

Authors	Pollack & Ringuette	Uhrmacher & Gugler	Choy et al.	Lees et al.	Xiong et al.
Year	1990	2000	2004	2007	2011
Language	Ada	Java	C++	C/C++	Java
Kernel	No	JAMES	DARBS	SIM_AGENT	SASSY
# Agent	1	1-2	1-16	1-64	1-1000
# Processors	1	1-2	1-16	1-16	1-16

1. TileWorld is essentially a simple environment, but sufficiently interesting to draw conclusions from an experiment. A simple environment is always a good starting place for initial experimentation. It makes the problem smaller and easier to evaluate.

2. TileWorld is highly parameterized. The experimenter can alter various aspects of the environment, for example, change the rate at which objects appear and disappear. This makes it possible to tune the experiments to examine particular aspects of interest.

3. TileWorld is well-understood. It is possible to compare results obtained from similar experiments using the same environment. This is important when experiments are in the early stages. [44].

Since its first implementation in 1990, TileWorld has been used as testbed for various purposes and modified to run on different simulation kernels. Table 5.1 summarizes the main features of 5 representative implementations of TileWorld.

Table 5.1 identifies, for each study, the year the paper was written or the TileWorld was produced, the language in which the TileWorld was written, the simulation kernel used and the numbers of agents and processors in the experiments.

The TileWorld by Pollack and Ringuette [63] is the seminal implementation. Though criticized by some as "too simple to study various features of the real world" [2], it was highly evaluated by others as “coming closest to satisfying the four properties an idealized simulated world should have” [42]:

1. A set of objects and events sufficiently rich to embody “interesting” aspects of real environments;

2. A metric of agent performance that is convenient to use, i.e., simple to calculate, yet sufficiently fine-grained to allow adequate discrimination of effectiveness;

3. A set of parameters that vary the interesting properties of the world. Ideally the parameters would map to well-defined measurable properties of real environments;

4. The ability to randomly generate large numbers of statistically similar worlds.

The TileWorld by Uhrmacher and Gugler [84] is a variant of the original TileWorld testbed. It includes “gas station” objects to top up the resources of the agents. In this variant, the agent’s resource-management skill is investigated by making each move consume fuel. Carrying a tile would cause the agent to consume more fuel. Therefore the agent would need to balance the consumption of resources with scoring points. This TileWorld runs on JAMES, which adopts the DEVS parallel simulator to exploit the parallelism inherent in the model. As does DEVS, Parallel DEVS [17] associates a simulator with each atomic model and a coordinator with each coupled model.

The TileWorld by Choy et al. [18] runs on the Distributed Algorithmic and Rule-based Blackboard System (DARBS) developed at the Open University and the Nottingham Trent University [59]. The blackboard system is analogous to a team of experts who communicate their ideas by writing them on a blackboard. The experts are represented by sets of rules, conventional procedures, neural networks, or other program modules. These modules are termed “knowledge sources” (KS). The blackboard is an area of global memory containing evolving information. The system’s current state of understanding of a problem is stored here as it develops from a set of data towards a conclusion.

The TileWorld by Lees et al. [47] runs on High Level Architecture (HLA) kernel that allows different simulations, referred to as "federates", to be combined into a single larger simulation known as a "federation". The federates may be written in different languages and may run on different machines. A federation is made up of 1) one or more federates; 2)a Federation Object Model (FOM); and 3) the Runtime Infrastructure (RTI).

JTileWorld is our production and we will devote the next section to it.

5.3 The JTileWorld

Our JTileWorld was built using the simulation logic of DARBS TileWorld mainly because

1. Accessible software: Dr. Hopgood and Dr. Choy, two authors of DARBS TileWorld, kindly provided us with their source code;

2. Complete documentation: The paper by Choy et al. [19] contains a complete list of the 31 rules that specify the behavior of the agents;

3. Rich and analytical/empirical results: The paper also provided data and statistical formulas for the results of their evaluation experiments.

As Figure 5.2 shows, DARBS TileWorld is under centralized control. The blackboard is the center of the simulation. All agents send their requests for new knowledge to the blackboard and get updated knowledge of the whole TileWorld from the blackboard. When an agent makes a move, it reports its new position, i.e., new knowledge, to the blackboard. The blackboard broadcasts the change to the "world", causing a "restart" to all the agents in the simulation.

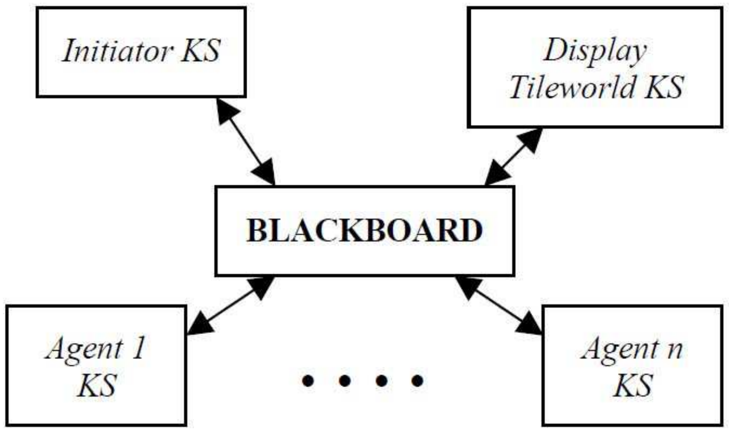
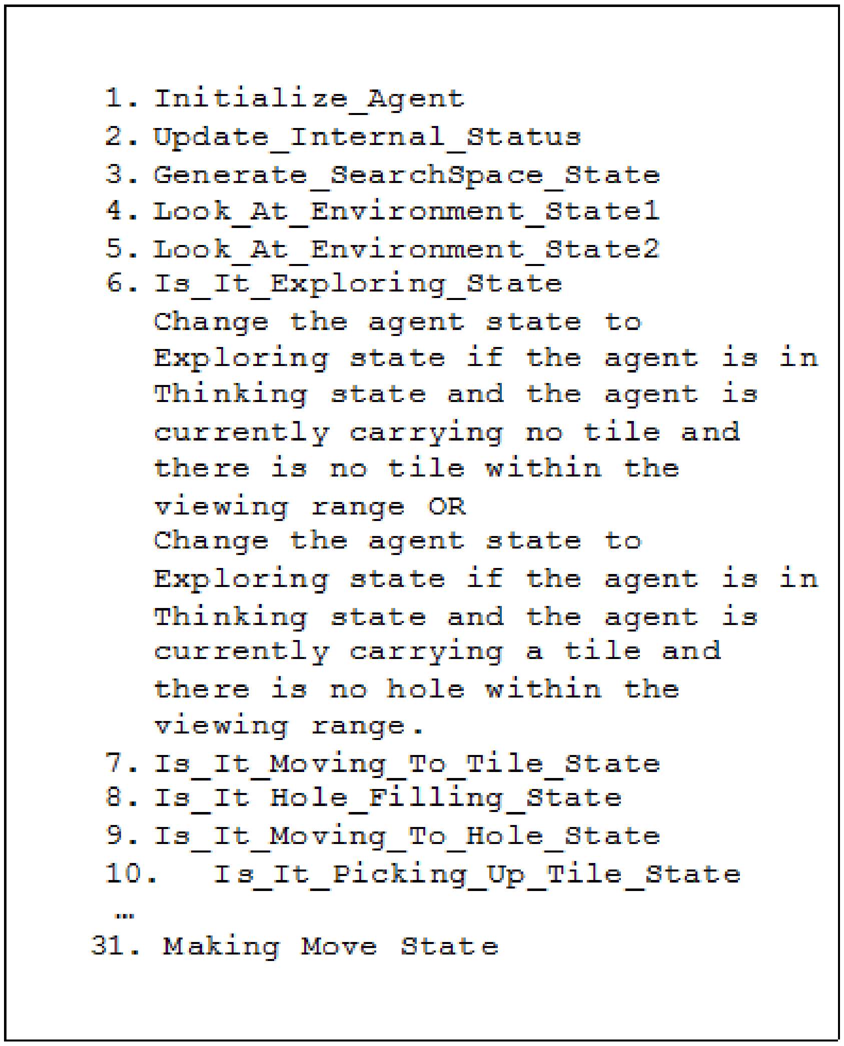


Figure 5.2: TileWorld Running on DARBS [18]

In DARBS TileWorld, agents $sense-think-act$ by a set of 31 rules shown in Figure 5.3. The 31 rules are stored as text files, one copy for each agent. At the beginning of a simulation run, each KS reads the rules from its own folder and stores the rules as its "knowledge". An agent "thinks" by rules, i.e., matching its current condition with one of the 31 conditions and then forming a plan according to the corresponding rule specified. The KSs fire all the rules even when certain rules are known beforehand to be dependent on other rules. The current implementation checks every sub-condition before evaluating whether the composite condition is true [18].



```
1. Initialize_Agent
2. Update_Internal_Status
3. Generate_SearchSpace_State
4. Look_At_Environment_State1
5. Look_At_Environment_State2
6. Is_It_Exploring_State
   Change the agent state to
   Exploring state if the agent is in
   Thinking state and the agent is
   currently carrying no tile and
   there is no tile within the
   viewing range OR
   Change the agent state to
   Exploring state if the agent is in
   Thinking state and the agent is
   currently carrying a tile and
   there is no hole within the
   viewing range.
7. Is_It_Moving_To_Tile_State
8. Is_It Hole_Filling_State
9. Is_It_Moving_To_Hole_State
10.   Is_It_Picking_Up_Tile_State
 …
31. Making Move State
```

Figure 5.3: Rules for Agents in DARBS TileWorld [18]

JTileWorld was built to run on SASSY, the PDES kernel for our cache-aware environment. We first built JTileWorld as a stand-alone simulation program, i.e., running on a single machine. The simulation logic specific to TileWorld was implemented and tested in this stage.

To make JTileWorld runnable on SASSY, we extended SASSY's abstract class logicalProcess in the way described in Chapter 3.

Figure 5.4 represents a 16 x 16 JTileWorld running on 4 WorkerPEs serviced by the MasterPE. This distribution is specified and performed by a “creator” provided by the application programmer. In this case, the 16 x 16 tile world is divided into 4 equal parts, each with 2 agents, 2 holes, 2 obstacles and 2 tiles. The 4 equal parts are then distributed to 4 WorkerPEs.

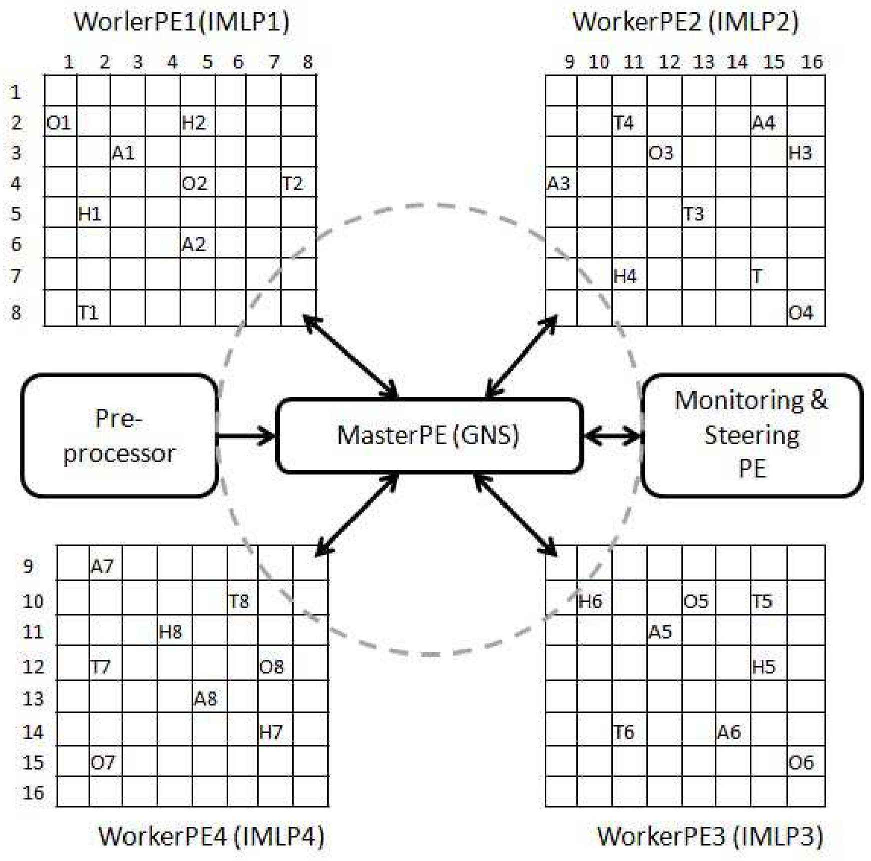


Figure 5.4: A 16 x 16 JTileWorld Running on SASSY

An agent first works in its own area on the WorkerPE where it “was born”, i.e., where it was initially assigned by a configuration file. When an agent finishes the work with its “hometown” and needs to explore another area of the world, its host WorkerPE communicates with the Global Name Server (GNS) to obtain the address (IP or machine name) of the destination and migrate the agent to that processor by sending a message to the WorkerPE residing on that processor.

In JTileWorld, the 31 rules are implemented as Java methods invoked by runLP() as illustrated by the pseudo code in Figure 4.5.

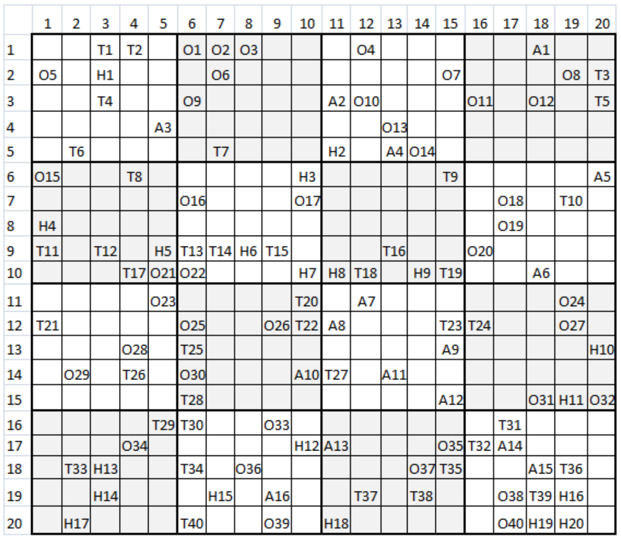
5.4 JTileWorld and DARBS TileWorld: A Comparison

DARBS TileWorld performed its evaluation experiments on an Ethernet 100Mbps switch network of personal computers (PCs). All the PCs used are AMD Athlon 1.67GHz processors with 224 megabytes (MB) of random access memory (RAM) running Red Hat 9 operating system with Linux kernel 2.4. A 20 x 20 TileWorld was created with 40 tiles, 20 holes and 40 obstacles. The position of the tiles, holes, obstacles and the initial positions of the agents were all randomly generated using the C++ standard random number generator function, rand() with a seed of 8. The number of active agents in the TileWorld varied from one to sixteen depending on the set-up. In the first set-up, DARBS TileWorld is run with one to sixteen agents on a single processor. In the second set-up, DARBS TileWorld is run with one to sixteen agents on multiprocessors (i.e., a new processor is added to the network for every new agent KS) [19].

In other words, the DARBS TileWorld is a 400-cell two-dimensional grid. Initially 4% of the 400 cells are occupied by Agents, 5% by Holes, 10% by Tiles and 10% by Obstacles. The rest of the world is empty. Figure 5.5 is a 20 x 20 tile world we generated using these parameters [19].

JTileWorld ran its experiments on the Linux cluster of the Computer Science Department at the University of Georgia. The Linux cluster is composed of 20 virtual Linux machines configured on a Sun Microsystems Sunfire X4600 filesever with 8 dual core AMD Opteron processors and a total of 32 GB RAM.

JTileWorld ran both single processor and multiple processor experiments using the setups described in [19]. We generated the 400-cell tile world using Java random number generator that generates algorithmic random number or pseudo random numbers, which are a fixed but random-looking sequence of numbers. The resulting tile worlds may not be exactly the same as those by DARBS TileWorld, but the proportion of agents, tiles, holes and obstacles are the same.

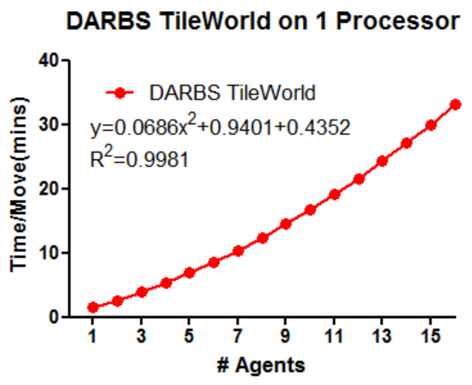


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1			T1	T2		O1	O2	O3				O4						A1		
2	O5		H1				O6								O7				O8	T3
3			T4			O9					A2	O10				O11		O12		T5
4					A3								O13							
5		T6					T7				H2		A4	O14						
6	O15			T8						H3					T9					A5
7						O16				O17							O18		T10	
8	H4																O19			
9	T11		T12		H5	T13	T14	H6	T15				T16			O20				
10				T17	O21	O22				H7	H8	T18		H9	T19			A6		
11					O23					T20		A7							O24	
12	T21					O25			O26	T22	A8				T23	T24			O27	
13				O28		T25									A9					H10
14		O29		T26		O30				A10	T27		A11							
15						T28									A12			O31	H11	O32
16					T29	T30			O33								T31			
17				O34						H12	A13				O35	T32	A14			
18		T33	H13			T34		O36						O37	T35			A15	T36	
19			H14				H15		A16			T37		T38			O38	T39	H16	
20		H17				T40			O39		H18						O40	H19	H20	

Figure 5.5: A 400-cell TileWorld with 16 Agents, 20 Holes, 40 Tiles and 40 Obstacles

For DARBS TileWorld, the basic performance metric is "time per move". An agent is considered to make a move when it has changed the TileWorld environment (i.e., moved to another cell, picked up a tile, or dropped a tile into a hole). Restarts due to other agents changing the TileWorld are not considered as moves. For every run of the experiment, each agent's average time per move is calculated over 50 moves. The overall average time per move for each run is then calculated as the average of all the agents' average time per move. The time per move was calculated by subtracting the time of an agent to move from the time of its subsequent move [19].

For the sake of comparison, JTileWorld used the same performance metric as DARBS TileWorld. Figure 5.6 displays the result for the single processor set-up.



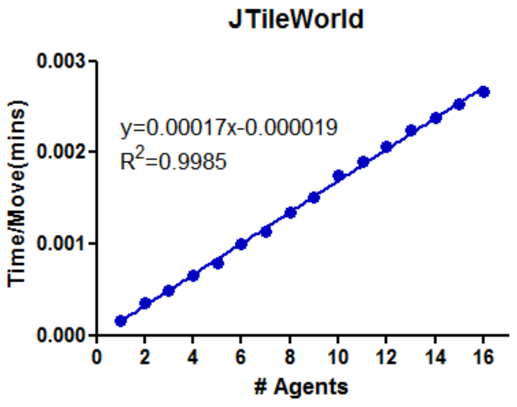


Figure 5.6: Average Time per Move on a Single Processor

Figure 5.6(a) presents the results of the single processor set-up for DARS TileWorld. The average time per move increases slightly more than linearly as the number of agent KSs increases. Choy et al. [19] claimed that the main cause is time-slicing: as the number of agent KSs increases, the single processor needs to time-slice between more processes and this takes up time [19].

Figure 5.6 displays the results by JTileWorld. A linear function is fitted onto the results to show the general trend. The main cause for the increase in average time per move is the lack of parallelism. When one agent is working, all the other agents have to "wait in idleness" because the processor can only run a single agent at any time. As a result, even if the total time to complete the tasks is approximately the same no matter how many agents are involved, the average time per move increases as the number of agents increases. For example, if one agent makes 40 moves in 40 seconds, the average time per move is 1 second. But if two agents make 20 moves each in 40 seconds, the average time per move becomes 2.

Similar trend can be observed in the multiple processor setting. In Figure 5.7(a), the line represents the time per move for DARBS TileWorld on a multi-processor setting. A linear function is fitted onto the results to show the general trend. Choy et al. [19] claimed that this is because on multiprocessors there is no time-slicing between the processes. The increase in average time per move is mainly due to the communication time between the processors.

The standard error of mean also increases as the number of agent KSs increases and this is due to the same reason as that on the single processor [19].

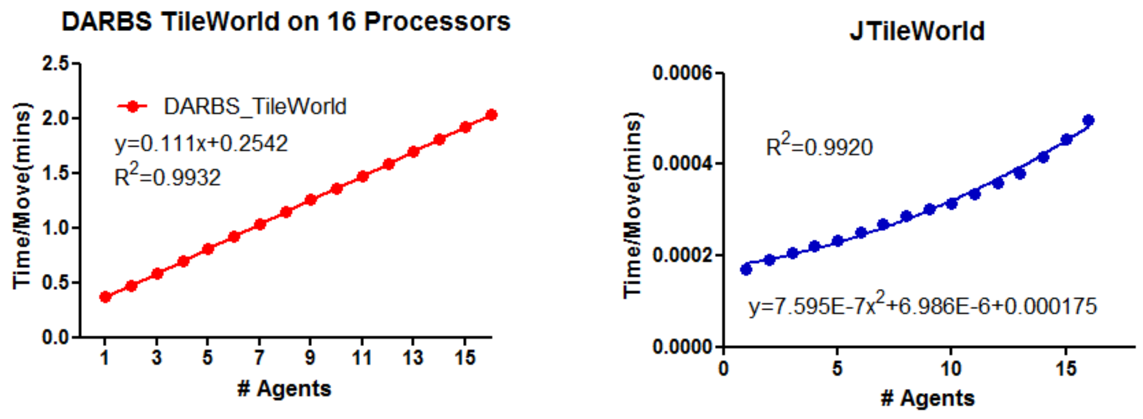


Figure 5.7: Average Time per Move on Multiple Processors

The result of JTileWorld for the multiple processor setting is displayed in Figure 5.7. A second order polynomial function is fitted to the result to show the general trend. The increase in average time per move is again due to the lack of parallelism. The 20 x 20 tile world is a small one. When the number of processors increases, the area each processor manages decreases, which means that the area each agent can roam freely without encountering other agents decreases. If two agents compete for the same resource, for example, a tile, a hole or an empty cell, one has to wait until the other retreats from the competition or consume the resource, in which case the waiting agent waited in vain and has to look for other resources.

The average time per move for JTileWorld is substantially shorter than DABRS TileWorld in both single-processor and multi-processor settings. The cause could be two-fold:

1. DARBS TileWorld is rule-based and the rules are read from text files and stored as character strings. Agents' "thinking" or "reasoning" is done by parsing the character-based rules that is time-consuming.

JTileWorld embedded the 31 rules into the runLP() method of Agent class. An agent does not need to parse character strings every time before it makes a decision on what to do next.

2. The Blackboard architecture is another cause for the slowness of DARBS TileWorld. The TileWorld is stored on the processor where the Blackboard resides. Whenever an agent modifies the TileWorld, it needs to report the change to the Blackboard, which causes all other agents to restart because the world has changed. The restart is expensive as it could force agents to stop and abandon what they have almost accomplished and begin from scratch.

   For JTileWorld, the TileWorld is distributed to multiple processors. Changes of one part of the TileWorld do not affect other parts of the TileWorld, which substantially reduces the number of global "restarts" and subsequently reduces the average time per move. The DNS-like architecture reduces the number of communications between agents and the MasterPE, which also contributes to the decrease of average time per move.

Chapter 6

Performance Evaluation

Caching efficiency depends on at least three factors: the cost of a cacheable computation, the number of such computations, and the caching overhead. In general, we expect better performance from caching as the cost of computation increases and as the costs of storing and retrieving results decreases. There are a few other issues to consider as well. At initialization time, the cache is empty and therefore not at all effective. However, as the cache "warms up", the performance improves. Accordingly, longer simulations are more likely to benefit from caching. The size of the cache is also important because for a given cache size, the number of key-value pairs stored is inversely proportional to the size of the key-value pairs. When the number of key-value pairs exceeds the cache size, either some of them will be cleared from the cache, or the cache size has to be increased, which means allocation of new memory space and a large amount of copying.

In our experiment, quantitative results were obtained using JPHold and JTileWorld.

JPHold is a Java version of the PHold application [29]. JPHold provides a synthetic workload using a fixed message population. Upon receiving a message, the LP schedules a new event whose destination LP is drawn from a uniform distribution ranging from 0 to one less than the number of LPs, which means that each LP is equally likely to be the destination of a message.

JTileWorld is a Java implementation of TileWorld which is a well-established test-bed for agent systems. For more details on TileWorld, please refer to the previous chapter.

6.1 Performance Evaluation Using JPHold

The experiments described in this section were run on UNIX Workstations (primarily SUN Ultra workstations) connected via Ethernet/Fast Ethernet to SUN Microsystems. Two types of experiments were performed: 1) Experiments to evaluate the role that pre-run statistics play in aiding decision making; and 2) Experiments to study the benefit of adaptive caching using statistics computed on-the fly.

Each of our experimental runs is defined by a set of parameters: the number of PEs (simulation schedulers), the number of Logical Processes (LPs), the message population, total events to be processed, the initial cache size, the load factor of the hash table, the computation granularity and more. For our experiments reported here, we used 10 machines that ran 40 PEs with a total of 1000 LPs evenly distributed over the 40 PEs. As workstations may have external loads and processes (not necessarily related to our simulation runs) while we ran our experiments we averaged the run time over all LPs to get the "mean time per event" which is then used in the speedup computation. For each setting, we ran the simulation 10 times and used the mean time in our reported results.

6.1.1 Pre-run Statistics Computation

To evaluate the overhead of caching, we used the Statistics Manager to collect and compute statistics of computation and caching on our workstations. The Statistics Manager uses a Fibonacci computation to measure the computation time. The first number of the Fibonacci sequence is 0, the second number is 1, and each subsequent number is equal to the sum of the previous two numbers of the sequence itself. The Fibonacci sequence has some qualities that suit measuring the caching overhead and the computation cost. First, it needs only 1 parameter so we can easily control the range of this parameter which, in turn, controls the cache hit rate; second, the time needed for the recursive computation of Fibonacci number covers a wide spectrum of time lengths, so we can generate workload of all kinds of granularities with the Fibonacci function; and third, it is easy to implement.

Table 6.1: Computation Costs

k	mean time(ms)	cumulative mean time(ms)
20	0.16	0.025
30	19.31	1.689
31	31.31	2.644
35	214.9	16.05
40	2370.3	155.7

We computed the mean computation time for different values of the input parameter k by running Fibonacci on a certain k 100 times and then compute the mean running time.

Table 6.1 contains the statistics of the Fibonacci function where k is the input parameter for the Fibonacci function. The “mean time” column shows the mean cost for computing Fibonacci numbers with a certain k. The last column contains the cumulative mean, which is the mean for the computation costs of Fibonacci sequence with k going from 1 to k, namely, the mean of $Fibonacci(1) + Fibonacci(2) + \ldots + Fibonacci(k)$ computation cost.

The “cumulative mean time” is used for our experiments with JPHold. The k's for “fibonacci” are drawn from a uniform distribution in the range from 1 to k. If “cumulative mean cost” for a certain range of k's is greater than the caching overhead, we set the “cache flag” to “on”.

To measure the caching overhead for “fibonacci”, we ran “cachingFibonacci” on a certain k 100 times. “cachingFibonacci” is a method provided by the Statistics Manager. It computes Fibonacci number, consults the cache and put the “key-value” pair into the cache each and every time no matter it is a cache hit or miss. By subtracting the mean computation time from the mean total time of running “cachingFibonacci”, we obtain the mean caching overhead.

For example, by running Fibonacci for $k = 20$ for 100 times, we obtained 0.16 ms as the mean computation cost. If the mean cost of running “cachingFibonacci ” 100 times on

k = 20 is 1.66 ms, then by subtracting 0.16 from 1.66, we obtained 1.5 ms as the mean caching overhead.

By measuring computation costs and caching overheads , we conclude that it is worthwhile to cache a function (or block) on SASSY when the granularity of computation is at least 1.5 ms (this is for 10 machines and the test environment described earlier).

6.1.2 Adaptive Caching Experiments: Hard Caching

With the pre-computed statistics presented in the previous section, we know that any computation with a granularity greater than 1.5 ms is a potential candidate for our caching scheme, i.e., turning on cache will potentially enhance performance.

As a test, we set the range of k to be 1-31 for the cacheable method "fibonacci" we implemented for JPHold. Figure 6.1 displays the speedup of cache-on over cache-off.

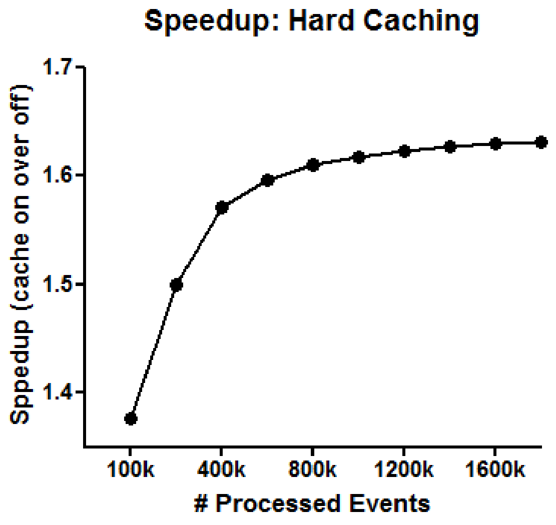


Figure 6.1: Speedup: Hard Caching for JPHold

With a mean caching overhead of 1.5 ms, consulting the cache when k is within the range of 1-25 will not enhance the performance because the computation time is shorter than the time of consulting cache. Only when k is bigger than 25, i.e., the computation cost is greater than 1.5 ms, the benefit of caching can offset the overhead of caching. But "hard caching" is set before the simulation begins and will consult the cache no matter what range k is in.

6.1.3 Adaptive Caching Experiments: Soft Caching

Relying on pre-computed statistics is appealing because it is easy to use and the performance enhancement is guaranteed if the computation granularity can be accurately computed beforehand. For simple computations, especially those driven by random numbers, if we know the distribution of the random numbers, we can use our Statistics Manager to obtain computation granularities in advance. But for computations that involve parameters whose distributions are unknown beforehand, it is hard to compute statistics for their computation granularities without running the simulation.

If the cache flag is set to "3" (i.e., soft caching), our Statistics Manager collects statistics while the simulation is running. It computes (and re-computes) statistics on-the-fly and makes decisions as to whether the cache should be turned on or off for a certain cacheable computation.

To test the effectiveness of the on-the-fly decision making, we set the range of k to be from 1 to 31 and then run JPHold with "cache off", "hard caching" and "soft caching" respectively.

For "cache off", $fibonacci$ method is invoked each time no matter what value k is assigned to. Apparently numerous computations are repetitive and redundant as the range of k is small.

For "hard caching", since the pre-computed cumulative mean time is 2.644 ms as shown in Table 6.1, which is larger than the caching overhead of 1.5 ms, the cache flag is set to "on" before simulation begins and the cacheManager consults the cache for each and every k, which results in substantial caching overhead as the computation cost for any $k < 20$ is less than caching overhead.

With “soft caching”, the Statistics Manager is invoked. It starts by gathering information about the cost of the computations for different values of k. After a while, it accumulates enough information to approximate the costs of the computation for different values of k. When it sees a specific k, it first finds out the approximate computation cost of that k and compares the cost with the threshold, in this case, the pre-computed caching overhead. If the cost is greater than the threshold, it consults the cache, otherwise, it yields to the “fibonacci” method.

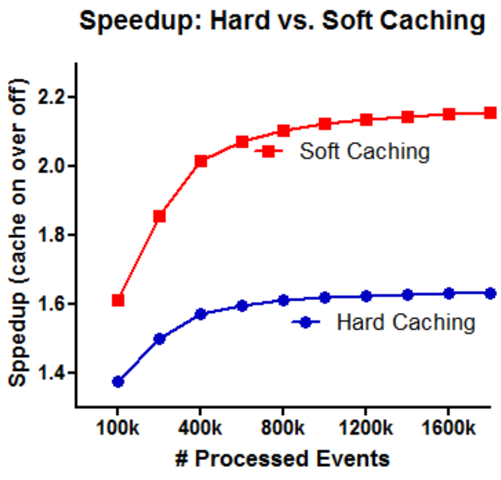


Figure 6.2: Speedup: Soft Caching for JPHold

Figure 6.2 shows the speedup of “hard caching” vs. “soft caching” over “cache off”. The blue (lower) line represents the speed up gained over cache-off by “hard caching”, i.e., cache is turned on at the beginning of the simulation (and does not change). The red (top) line represents “soft caching”, i.e., the cache is turned off for fine-granularity computations and on for coarse-granularity computations.

“Hard caching” and “soft caching” have their own favorite cases where one performs better than the other. For those computation blocks that mainly rely on input parameters whose distribution can be decided in advance, “hard caching” is more advantageous because by the help of the Statistics Manager we can easily find out its computation cost. But for computation blocks that involve parameters whose distribution relies on run-time situation, thus can hardly be determined before simulation begins, "soft caching" would be more advantageous because the Statistics Manager will "learn" from the changing situation.

6.2 Performance Evaluation using JTileWorld

With the ever-increasing speed of CPUs, it becomes more and more challenging to design smart caching algorithms with which the caching benefit can sufficiently offset the caching overhead. In many occasions, the time to access the "cache" can be disproportionally longer than simply doing the computation itself.

We have shown that only when the mean cost of agents' $sense-think-act$ cycle is higher than 1.5 ms can the caching benefit surpass the caching overhead for our caching scheme. The simulation has to be coarse-grained. But for PDES benchmark programs such as PHold, the LPs do not need $sense - think - act$. They are reactive rather than deliberative and their reaction time can be much shorter than accessing the cache.

When using JPhold for experiments on our Computation Block Caching algorithm, we deliberately added a "penalty function" to imitate a $sense - think - act$ cycle typical to deliberative agents. $fibonacci$ serves as the $penaltyfunction$ for JPhold. That is to say, each time before an LP sends out a message, it invokes the $fibonacci$ function, computing the kth number of Fibonacci number.

With the "penalty function", JPHold was successfully used in proving the basic concepts of Computation Block Caching, but a key question remains: is there any real-world simulation program that can benefit from our "Computation-block Caching" scheme?

The answer is "Yes!". The proof is JTileWorld.

TileWorld is a real-world simulation application and our experiment with JTileWorld has proven that the benefit of caching using the Computation Block Caching scheme can substantially offset the overhead of caching, enhance the performance of the simulation and make the simulation more scalable.

We designated 1 computation block in JTileWorld as “cacheable”. It is within the runLP() method and covers the basic simulation logic of JTileWorld. As Figure 4.5 shows, this is the “sense-think-act” block that makes a change to the TileWorld. An agent either picks up a tile, fills a hole or moves to another cell after this block of code is carried out.

The “sense-think-act” cycle can take substantial time to fulfill. For DARBS TileWorld, the average time per move can take as long as 33 minutes [19]. For the TileWorld by Lees et al., the average time per cycle can take as long as 0.5 second [47]. For JTileWorld, the “sense-think-act” cycle can take as long as 0.3 second.

In this sense, caching the “sense-think-act” cycle has great potential to enhance the performance of TileWorld.

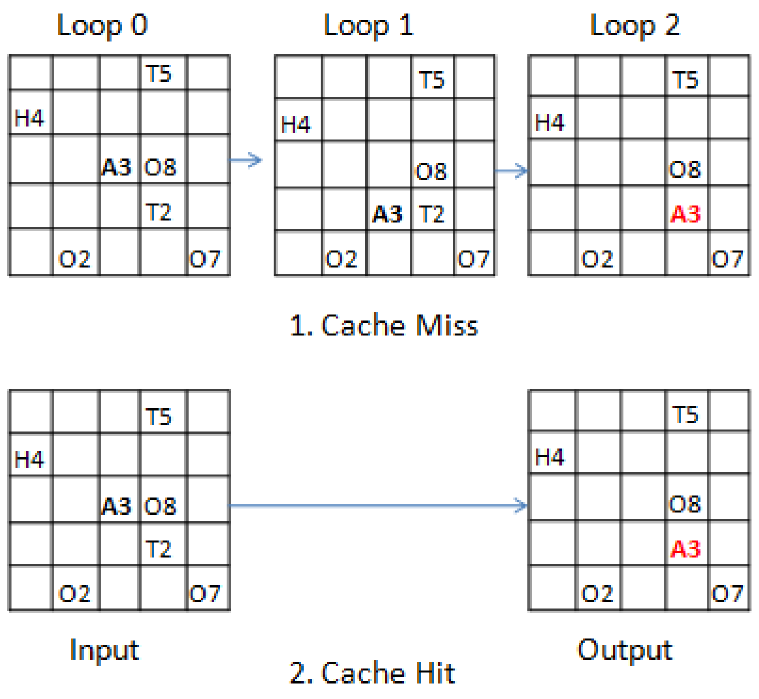


Figure 6.3: Cache Miss vs. Cache Hit for Computation Block “getTile”

Figure 6.3 illustrates the “cache miss” and “cache hit” cases for the computation block “getTile”. Here A3 is the agent who is looking for tiles in its 5 x 5 viewing area. Before A3 enters the “getTile” block, its initial position is as Loop0 shows. If the “cachingFlag” is set to "on" for this computation block, the CacheManager would begin to work by consulting the cache with package name, class name, block name and the 2 variables as input parameters. In this case, "tileInHand" is "false", "myLastRegion" is the 5 x 5 region shown in Loop0.

If the cache consultation results in a "cache miss", the CacheManager yields to the application code and waits until A3 comes out of the "getTile" block. It then generates a hash key with the region in Loop0 along with the package name etc. and caches the key with the region in Loop2 and "tileInHand = false". Next time, if the same situation is encountered, the cache consultation will result in a "cache hit" and the agent could skip the "getTile" block by jumping to Loop2.

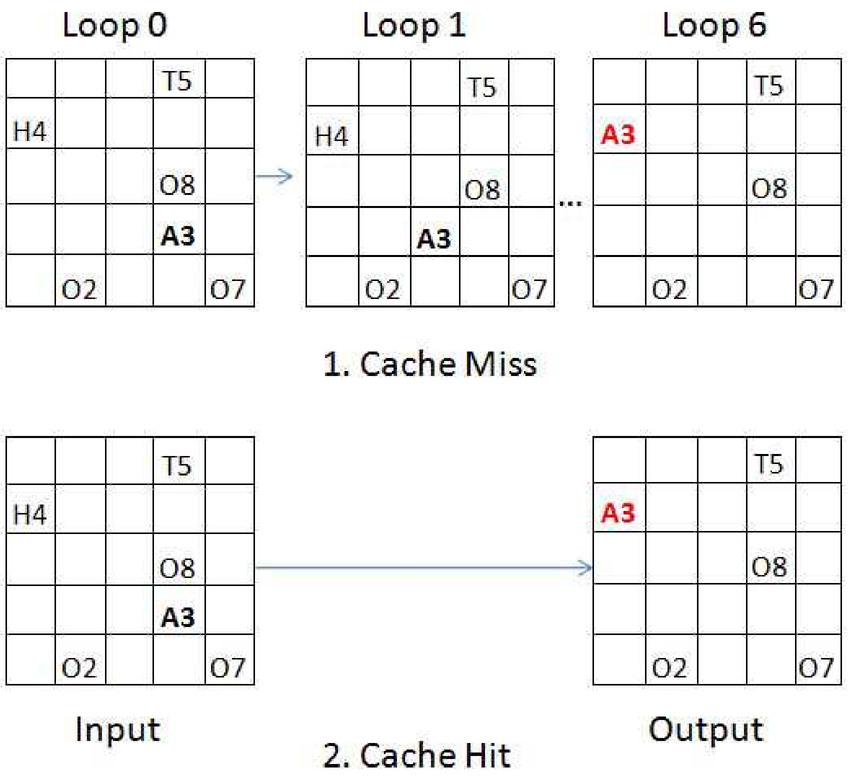


Figure 6.4: Cache Miss vs. Cache Hit for Computation Block "fillHole"

Figure 6.4 illustrates the "cache miss" and "cache hit" cases for the "fillHole" block. The mechanism of caching is similar to that of "getTile" described above.

But the caching overhead is also non-negligible as the input parameters and the return variables are both complex. To cache a “key-value” pair, the CacheManager needs to gather the current values of the variables, concatenate the values with package name, class name, and computation block name, then invoke Java String’s hashCode() method to get the hash key. Depending on the quantity and complexity of the input parameters and state variables, the caching overheads could range from a fraction of a millisecond to tens of milliseconds. For the set-up described here, the caching overhead is approximately 2.5 milliseconds.

In addition, there is redundant and even “harmful” information in the input parameters which could dramatically reduce the rate of cache hits. For example, the 2 obstacles $O2$ and $O7$ at the bottom of the viewing area do not block $A3$’s way to pick up the tile or fill the hole. But as a constituent member of the “myLastRegion”, they are also passed in as the input parameter. If a similar topography is encountered with some minor difference such as no obstacles at the bottom of “myLastRegion”, a cache consultation would result in a cache miss rather than a cache hit, which is counter-intuitive because visual knowledge and common sense tell us that without two obstacles at the bottom, $A3$ would be happier in its journey to fill the hole of H4.

6.2.1 Impact of Caching on Scalability

Scalability refers to the ability of a system to either handle growing amounts of work in an efficient manner, or to be readily enlarged. Scalability can be further categorized as vertical scalability and horizontal scalability. Vertical scalability increases the resources of the elements of a system. Horizontal scalability is achieved by adding more elements to a system.

Previous TileWorlds suffer from the lack of scalability. For example, when the number of agents increases to 16, the time per move on DARBS TileWorld (for a single processor setting) is about half an hour! DARBS does have some ability to scale. When it was horizontally scaled to use multiple machines, the time per move substantially decreased to 2 minutes (for 16-processor setting)[19].

Choy et al. didn't report the time per move for more than 16 agents, but they fitted a second order polynomial function $y = 0.0686x^2 + 0.9401x + 0.4352$ to the results of their single-machine setting[19]. Substituting x with 100 and 1000 into the function, we obtained $y = 780$ minutes and $y = 69540$ minutes respectively. This means that if the number of agents increases to 100 and 1000, the time per move would be about 780 minutes and 69540 minutes respectively.

When scaled horizontally, DARBS TileWorld scales better than a single processor. A linear function $y = 0.111x + 0.2542$ can be fitted the results of their multiple-machine setting[19]. Substituting x with 100 and 1000 into the function, we obtained $y = 11$ and $y = 111$ respectively. This means that if the number of agents increases to 100 and 1000, the time per move would be about 11 minutes and 111 minutes respectively.

Lees et al. did not report any functions fitted to their results, but using the information available in their paper [47] (as shown in Figure 6.5), we fitted a linear function $y = 0.7796x - 0.0757$ onto their results for the single federate case (as shown in Figure 6.6. The R^2 is 0.9984, indicating an almost perfect fit.

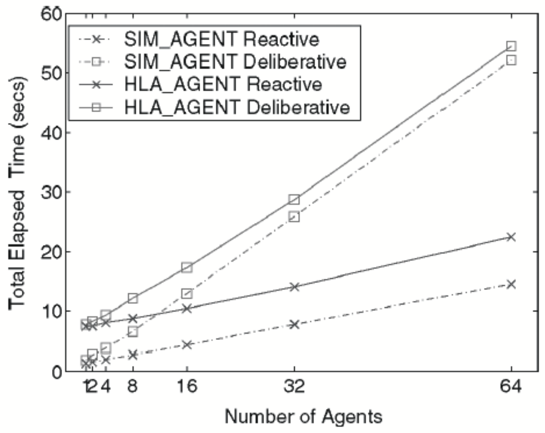


Figure 6.5: Total elapsed times for 1-64 reactive and deliberative agents in SIM_AGENT and HLA_AGENT(single federate)

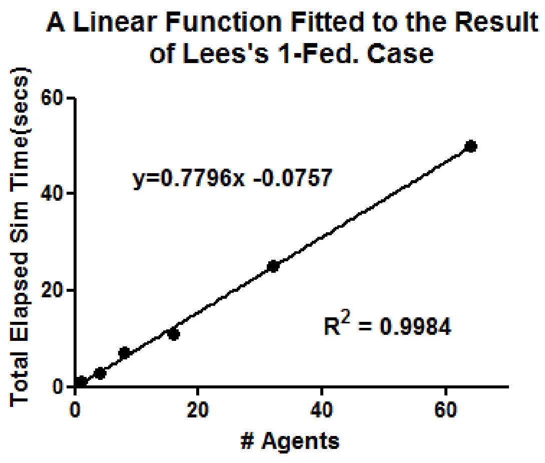


Figure 6.6: A linear function fitted to the result of the single-federate case in [47])

A linear function with a positive coefficient for x indicates a steady increase of y when x increases. Substituting x with 100 and 1000 into the function, we obtained $y = 78$ and $y = 780$ respectively, which means that for a single-federate HLA_AGENT, the total elapsed time for 100 and 1000 agents would be about 78 seconds and 780 seconds respectively.

Note that the time reported by Lees et al.[47] is not the time per move, but the total time elapsed for 100 simulation cycles. The total elapsed times for each simulation cycle for the distributed their experiments can be further broken down into the time for the simulation phase (running the user simulation plus object registration and deletion, attribute ownership transfer requests, and queuing attribute updates for propagation at the end of the user simulation cycle) and the RTI phase (flushing queued attribute updates to the RTI, applying incoming attribute updates to the slots of local objects and proxies, processing object discoveries and deletions, and synchronizing with other federates)[47].

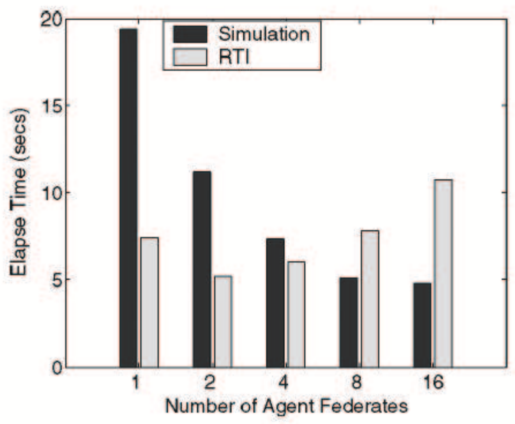


Figure 6.7: Simulation and RTI phase times for an Agent Federate(64 Reactive agents distributed over 1-16 Nodes)[47]

As Figure 6.7 shows, the total elapsed time not only included the simulation time, but also the RTI time. In addition to these, it also added a 10 ms "deliberation penalty" for each plan generated, and the agents replanned whenever the region of the Tileworld they could sense was changed by the actions of another agent or by the environment itself. The 10 ms is towards the lower end of the deliberation times reported in the multi-agent system(MAS) literature. For example, the results reported in [70] are for agents with cycle times in the range 95-105 milliseconds, and [74] report experiments with planning agents that require from 2 seconds to 20 hours of CPU time per cycle [47].

We also fitted a function onto their results of the 16-federate case. It is a second order polynomial function $y = 0.368x^2 - 8.226x + 56.1$ with $R^2 = 0.8779$.

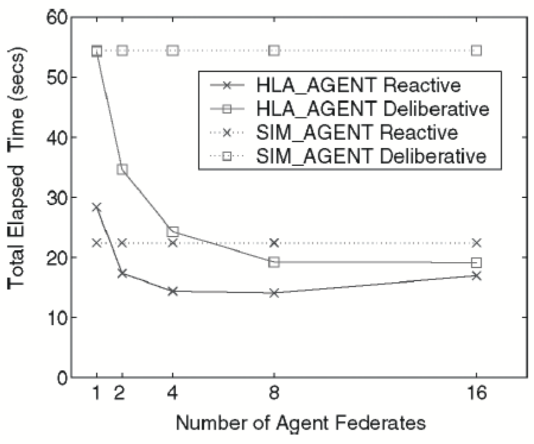


Figure 6.8: Total elapsed time for an Agent Federate (64 Reactive and Deliberative Agents distributed over 1-16 nodes)

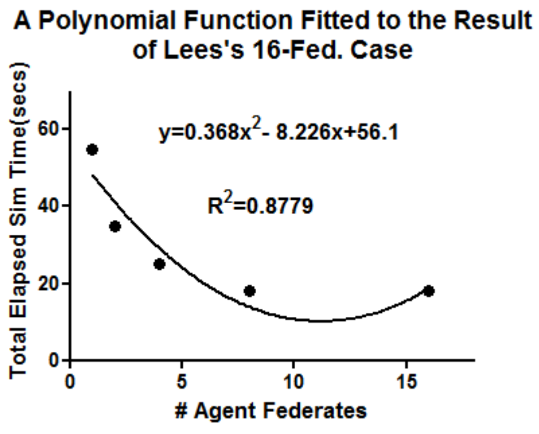


Figure 6.9: A polynomial function fitted to the result of the multiple-federate case in [47])

As Figure 6.8 shows, the shortest total elapsed time was reached when the numbers of federates are 32 and 64. Adding more federates to the simulation would not enhance the performance much further, rather, it may negatively affect the performance.

Substituting x with 100 and 1000 into the function, we obtained $y = 168$ and $y = 1037$ respectively, which means that for a multiple-federate HLA_AGENT, the total elapsed time for 100 and 1000 agents would be about 168 seconds and 1037 seconds respectively.

Of course, this can happen only when all the conditions do not change, i.e., a Tileworld of 50 units by 50 units with an object creation probability (for tiles, holes, and obstacles) of 1.0 and an average object lifetime of 100 cycles: objects are created and destroyed at approximately the same rate. The Tileworld initially contains 100 tiles, 100 holes and 100 obstacles.

To test how Computation Block Caching would impact scalability, we performed experiments with JTileWorld using the settings described in Lees et al[47]. with one parameter space (the number of agents) changed to serve our purpose.

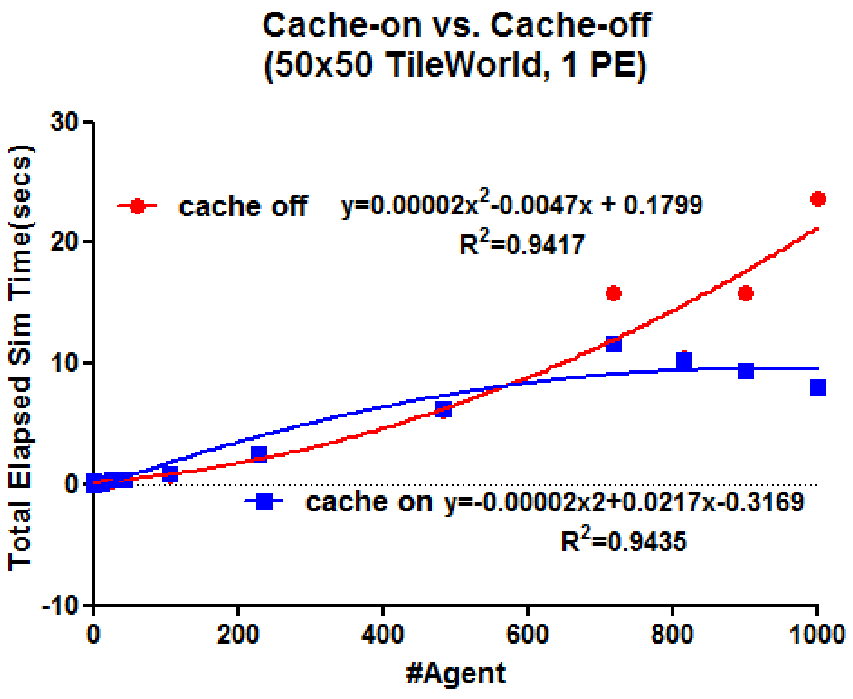


Figure 6.10: Total elapsed times for 1-1000 deliberative agents in JTileWorld

In Figure 6.10, the x coordinate shows the number of agents, ranging from 1 to 1000. The y coordinate shows the total elapsed time for 100 simulation cycles. Except for the start-up time (creating JTileWorld and distributing it to PEs) and shut-down time(computing and reporting simulation statistics), all other times were included in the “total elapsed time”.

Two polynomial functions were fitted to the results of cache-on and cache-off for the 1-PE case. As number of agents increased, the time per move increased for cache-off scenario. It is mainly the result of time slicing as the single processor had to swap among the agents. For cache-on case, the time slicing still existed, but with each cache hit, one or more moves were skipped, thus saving some time of $sense - think - act$ cycles, for time slicing and for management activities performed by PEs.

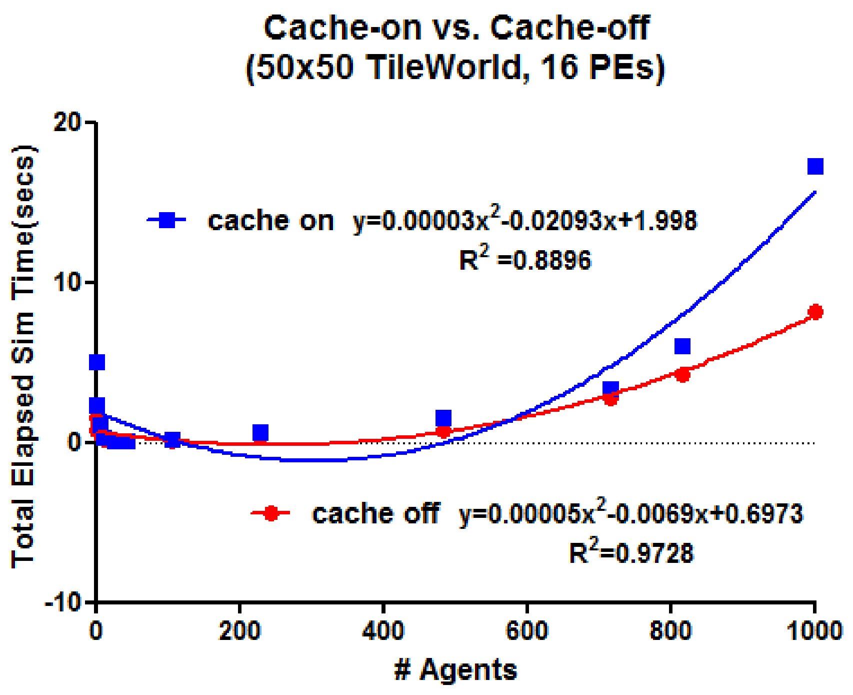


Figure 6.11: Total elapsed times for 1-1000 deliberative agents in JTileWorld distributed over 1-16 PEs

Figure 6.11 contains the result for the 16-PE case. This time, we see two polynomial curves going to the same direction with cache-on curve similar to cache-off in the middle section while higher on two ends. This is preconditioned by the topology, i.e., the size of the TileWorld and how it is distributed to the PEs.

When only 1 agent worked on the TileWorld, it could only be assigned to 1 PE, resulting in all the other 15 PEs idling away their time. As the number of agents increased, all the

Table 6.2: Scalability Comparison: DARBS TileWorld(DTW), HLA TileWorld(HTW) and JTileWorld(JTW)

TileWorld	Function	x=100, y=?	x=1000, y=?
DTW Single	$y = 0.0686x^2 + 0.9401x + 0.4352$	780(min)	69540(min)
HTW Single	$y = 0.7796x - 0.0757$	78(sec)	780(sec)
JTW Single(cache off)	$y = 0.00002x^2 - 0.0047x + 0.1799$	0.7(sec)	23(sec)
JTW Single(cache on)	$y = -0.00002x^2 + 0.0217 - 0.3169$	0.9(sec)	8(sec)
DTW Multiple	$y = 0.111x + 0.2542$	11(min)	111(min)
HTW Multiple	$y = 0.368x^2 - 8.226x + 56.1$	168(sec)	1037(sec)
JTW Multiple(cache off)	$y = 0.00005x^2 - 0.0069x + 0.697$	0.1(sec)	8(sec)
JTW Multiple(cache on)	$y = 0.00003x^2 - 0.02093x + 1.998$	0.2(sec)	17.3(sec)

PEs would finally have some agents working on their regions and contributing to the total number of moves. When the number of agents further increased, more and more agents were assigned to each PE, which became more and more crowded. With 1000 agents on the 50 x 50 TileWorld distributed to 16 machines, the mean number of cells for each agent is fewer than 3. The agents were bounded by one another and had little space to move about!

But agents would still try to perform their job. They continued on $sense - think - act$ only to find that there are nowhere to go. In this case, cache-on would negatively affect the performance because many times, a cache hit would result in no movement as the agent had nowhere to move.

Table 6.2 summarizes the scalability of the 3 TileWorlds we discussed above.

6.2.2 Impact of Cache Hit Rate on Performance

We evaluated the impact of cache hit rate on performance of our cache-aware middleware by "time per hole filled", computed using the following equation:

$$time\,per\,holes\,filled = (total\,time/total\,holes\,filled)/total \tag{6.1}$$

Here “total time” and “total holes filled” refer to the mean total time and mean total holes filled respectively over 10 runs of the same simulation setting.

The computation costs of these two blocks and the caching overhead are decided by a number of factors including the size of the TileWorld, the number of PEs, the number of LPs, none of which change on the fly, so we only tested “hard caching” on JTileWorld.

This is a continuation of the experiments reported in the last section, so we retained most of the parameters except for the size of the tile world. The tile world used for this experiment was set up as the following:

1. size of the TileWorld: 400 x 400
2. number of PEs: 16
3. number of agents: 1000
4. number of tiles: 6400
5. number of holes: 6400
6. number of obstacles: 6400

We ran the experiments with a certain setting for both cache-on and cache-off options. That is, for each run, we controlled for all other parameters except for the cache-on/cache-off parameter.

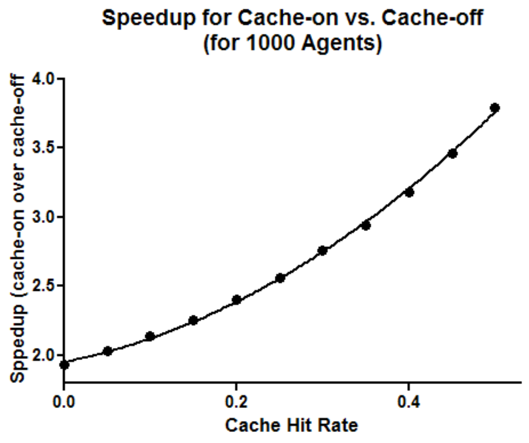


Figure 6.12: Impact of Cache Hit Rate on the Performance of JTileWorld

As Figure 6.12 indicates, for a fixed setting and a stable computation block (by "stable" we mean that the computation time does not change drastically), the most important factor controlling performance would be cache hit rate. In order to test the impact of cache hit rate on performance, we designed a series of tile worlds with different cache hit rates ranging from 0 to 50% and tested our caching scheme on these worlds.

When cache hit rate is 0, there is no speedup. Instead, the "time per hole filled" is higher for cache-on than for cache-off. When cache hit rate increases to about 5%, the performance of cache-on begins to catch up with cache-off. When cache hit rate reaches 15%, the performance of cache-on definitely surpasses that of cache-off.

We also investigated the impact of different numbers of PEs have on performance. These experiments were conducted on the CF Cluster of Computer Science Department at the University of Georgia. The CF Cluster is composed of 8 nodes, each of which is a Dell Poweredge R210 computer with Quad core 2.4 GHz Xeon processors and 4 GB Ram.

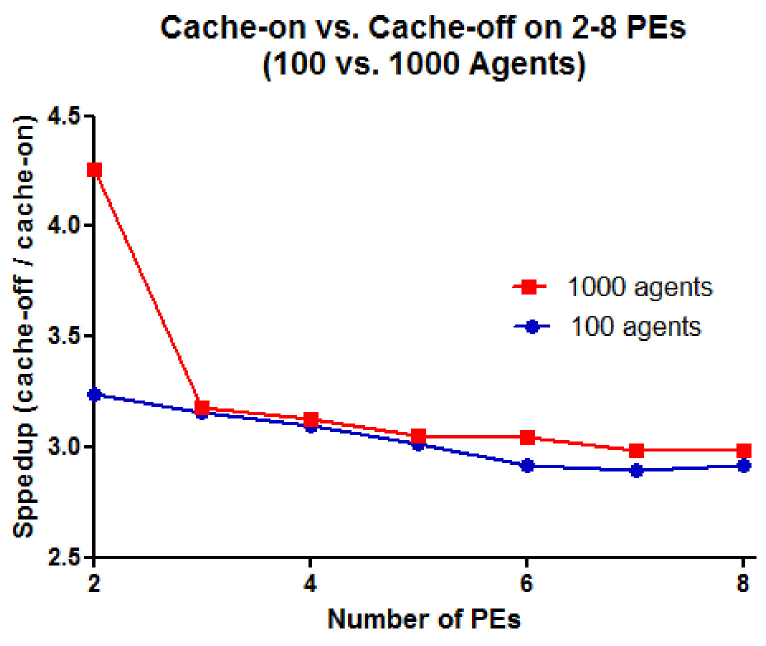


Figure 6.13: Speedup for Cache-on vs. Cache-off over 2-8 PEs

Figure 6.13 shows the speedup of cache-on vs. cache-off for 100 and 1000 agents on 2 to 8 PEs. The top line represents the speedup by 1000 agents. The most drastic speedup is gained with the 2-PE setting, i.e., one PE for the MasterPE and the other for the WorkerPE. The bottom line represents the speedup by 100 agents. The highest speed is also obtained with the 2-PE setting. With the increase of the number of PEs, the size of the TileWorld distributed to each PE decreases. As all LPs on the same PE share the same "cache", the increase of PEs results in lower cache hit rate. Subsequently the speedup levels out. On the average, cache-on runs about 3 times faster than cache-off, which proves that our caching mechanism is beneficial to performance.

6.2.3 Impact of Caching on Rollback

SASSY is a PDES kernel that allows out-of-order executions. It is up to the simulation user to decide whether or not to run their simulation in a Time Warp fashion, i.e., whether allowing different parts of the simulation to advance at their own speed and remedy any inconsistencies by rollback. Rollback can be turned on or off by specifying the value of $rollbackFlag$ in the configuration file.

In our experiments reported in previous sections, rollback was turned off in order to focus on the impact of cache hit rate on performance. Now we are going to take rollback into consideration.

An optimistic PDES consists of a collection of logical processes (LPs) that communicate by exchanging timestamped messages. To ensure correctness, each LP must achieve the same result as if incoming messages were processed in timestamp order. If an LP receives a "straggler" message with timestamp smaller than that of others already processed by the LP, event computations with timestamp larger than the straggler are rolled back, and reprocessed (after the straggler) in timestamp order. Each message sent by a rolled back computation is cancelled by sending an anti-message that "annihilates" the original message [29].

In order not to rollback to the beginning of the simulation each time an error occurs, a safe time point must be computed so that all LPs only need to rollback to that point. This safe point is called Global Virtual Time (GVT). By definition [29], GVT(T) is the minimum timestamp of any unprocessed messages or anti-messages in the system at real time T. It defines a lower bound on the timestamp of any future rollbacks [29].

JTileWorld consists of "time zones" assigned to different WorkerPEs, each with its own "local time", i.e., Local Virtual Time (LVT). An agent proceeds by sending timestamped messages to itself. These messages are processed by the local WorkerPE in timestamp order, which guarantees no out-of-order execution by the same WorkerPE.

When an agent has finished up with its work in its own "time zone", it would need to migrate to another "time zone". If the $rollbackFlag$ is set to 'ON, a migration will cause a recomputation of GVT. The MasterPE broadcasts a "GVT recomputation" message to all the WorkerPEs in the simulation, who then send in their LVT to the GNS. The MasterPE broadcasts the minimum LVT as the new GVT to all the WorkerPEs, who, upon receiving the rollback command, rollback to the new GVT. Any resources before the new GVT can be released.

To evaluate the impact of caching on simulations with rollback, we controlled for all the other parameters except the $cachingFlag$ variable that can be set as ON or OFF.

For these experiments, in order to make the result comparable with the result in previous section, we used the same simulation set-up and same performance metrics.

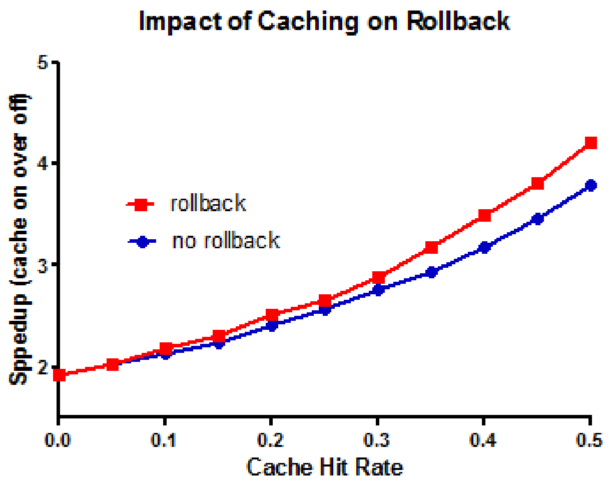


Figure 6.14: Impact of Caching on Rollback

As Figure 6.14 displays, we gained larger speedup of cache-on over cache-off when rollback is present. The speedup is still dominated by cache hit rate, but rollback added more chances for cache hits because rollback leads to a partial re-run of the simulation. When agent number was 1, there was no rollback as the agent's timestamp is both the LVT and the GVT. When agent number increased to 2, there was a possibility of rollback, but not high. As the number of agents increases, the probability of rollback also increases, and cache hit rate increases accordingly.

Chapter 7

Conclusions and Future Work

We designed and implemented SASSY, the Scalable Agents Simulation System that integrates Agent-Based Simulation(ABS) with Parallel/Distributed Discrete Event Simulation(PDES). We are not the first to do so, but our implementation uses the most up-to-date technologies in both ABS and PDES and provides user-friendly APIs for ABS programmers as well as PDES programmers.

In order to enhance the performance of PDES for ABS, we designed and developed a cache-aware middleware with an innovative Computation Block Caching scheme. We experimentally proved its merits in applicability and performance.

Our caching scheme tackled two major problems that traditional caching schemes had not overcome yet, namely, the dependencies of state variables and the return of multiple results. A computation block is not limited to a function (or a method). It can be any chunk of code. Computation Block Caching does not require recoding either on the application side or on the kernel side. It sits between the two and integrates the two into a seamless whole.

We designed and developed a Preprocessor that reads the application-provided specifications and generates a cacheable version for each specified computation block. The specification for cacheable computation blocks can be modified any time as needed. The Preprocessor is invoked only when modifications are made to the specifications. Further, the caching scheme is adaptive in the sense that the cache can be turned on and off for each individual cacheable computation block according to statistics gathered beforehand or on-the-fly, and is applicable to simulations with variable degree of being reactive and deliberate. We provided a Statistics Manager to facilitate both hard caching and soft caching.

In order to accommodate ABS involving “situated agents” and test SASSY’s API for ABS programmers, we designed and developed JTileWorld, a Java version of TileWorld in which both the agent and the environment are highly parameterized, enabling one to control certain characteristics of each. Users can experimentally investigate the behavior of various meta-level reasoning strategies by tuning the parameters of the agent, and can assess the success of alternative strategies in different environments by tuning the environmental parameters.

We tested our cache-aware environment on JPHold, a Java version of the PDES benchmark program, and JTileWorld, a Java version of TileWorld involving “situated agents”. We experimentally proved that Computation Block Caching enhances performance and scalability. We also empirically proved that caching performance is dominated by computation granularity while also affected by many other factors including cache hit rate, parameter size and rollback rate.

Our future work includes designing better algorithms for JTileWorld, investigating and quantifying the correlation between a variety of parameters such as frequency of global restarts, seeds of random numbers, size of keys (as in key-value pair) with caching performance.

Bibliography

[1] M. Astley. (1999). The Actor Foundry. University of Illinois. http://osl.cs.uiuc.edu

[2] B., Al-Badr and S. Hanks. (1991). Critiquing the tileworld: Agent architectures, planning benchmarks, and experimental methodology. Technical Report TR 91-11-01.

[3] J.A.D.W. Anderson editor.(1989) POP-11 Comes of Age: The Advancement of an AI Programming Language. Ellis Horwood, Chichester.

[4] R. Bagrodia and W. T. Liao.(1994). Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Trans. Software Eng.*, 20(4): 225-238.

[5] T. Balch. (1998). Behavioral diversity in learning robot teams. Ph. D. thesis, College of Computing, Georgia Institute of Technology.

[6] T. R. Balch, J. Summet, D. S. Blank, D. Kumar, M. Guzdial, K. J. O'Hara, D. Walker, M. Sweat, G. Gupta, S. Tansley, J. Jackson, M. Gupta, M. N. Muhammad, S. Prashad, N. Eilbert and A. Gavin. (2008) Designing Personal Robots for Education: Hardware, Software, and Curriculum. *IEEE Pervasive Computing*, 7(2): 5-9.

[7] R. Barr,Z.J. Haas and R. Renesse. (2005). JiST: an efficient approach to simulation using virtual machines. *Softw., Pract. Exper.*, 35(6): 539-576.

[8] F. Bellifemine, A. Poggi, and G. Rimassa. (1999). JADE - A FIPAcompliant agent framework. *Proceedings of PAAM'99*, April 1999, pp. 97-108.

[9] F. Bellifemine, G. Caire, A. Poggi and G. Rimassa.(2008). JADE: A software framework for developing multi-agent applications. Lessons learned. *Information and Software Technology*, 50(1-2): 10-21.

[10] R. Beraldi, L. Nigro and A. Orlando.(2003) Temporal Uncertainty Time Warp: An Implementation Based on Java and ActorFoundry. *Simulation*, 79(10): 581-597.

[11] P. Bizarro, L. Silva and J. G. Silva. (1998). JWarp: a Java library for parallel discrete-event simulations. *Concurrency: Pract. Exper.*, Vol. 10(11-3), 999-005.

[12] E. Bonabeau. (2002). Agent-Based Modelling: Methods and Techniques for Simulating Human Systems. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 99(3): 7280-7287.

[13] R. E. Bryant. (1077) Simulation of Packet Communications Architecture Computer Systems. *MITLCS-TR-188, Massachusetts Institute of Technology.*

[14] C. J. E. Castle and A. T. Crooks. (2006). Principles and Concepts of Agent-Based Modelling for Developing Geospatial Simulations. Center for Advanced Spatial Analysis (University College London), Working Paper 110, London.

[15] K. M. Chandy and J. Misra. (1979). Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440:452, Sept. 1979.

[16] K. M. Chandy and J. Misra. (1981). Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Commun. ACM 24(4)*: 198-206.

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

[18] K. W. Choy, A. A. Hopgood, L. Nolle and B. C. O'Neill. (2004). Implementation of a tileworld testbed on a distributed blackboard system. *18th European Simulation Multiconference (ESM2004)*, Magdeburg, Germany, June 2004, Horton, G., (Ed.), pp. 129-135.

[19] K. W. Choy, A. A. Hopgood, L. Nolle and B. C. O'Neill.(2005). Performance of a multi-agent simulation on a distributed blackboard system. *Int. Journal of Simulation Systems, Science and Technology 6.* 57-72. ISSN: 1473-8031.

[20] A. Chugh and M. Hybinette. (2004). Towards adaptive caching for parallel and discrete event simulation. *WSC 2004: Proceedings of the 36th conference on Winter simulation.*

[21] J. S. Dahmann, R. Fujimoto, R. M. Weatherly. (1997). The Department of Defense High Level Architecture. *Winter Simulation Conference 1997*, 142-149.

[22] S. R. Das, R. Fujimoto, K. S. Panesar, D. Allison and M. Hybinette.(1994). GTW: a time warp system for shared memory multiprocessors. *Winter Simulation Conference 1994*, 1332-1339.

[23] J. Eberhard and A. Tripathi. (2007). Mechanisms for object caching in distributed applications using Java RMI. *Softw., Pract. Exper.*, 37(8): 799-831.

[24] J. M. Epstein. (1999). Agent-Based Computational Models and Generative Social Scienc. *Complexity*, 4(5): 41-60.

[25] R. Ewald, C. Dan, T. Oguara, G. Theodoropoulos, M. Lees, B. Logan and A. M. Uhrmacher. (2006). Performance Analysis of Shared Data Access Algorithms for Distributed Simulation of MAS. *Proceedings of the Twentieth ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation(PADS).*

[26] S. L. Ferenci, R. M. Fujimoto, M. H. Ammar, K. S. Perumalla and G. F. Riley.(2002)Updateable simulation of communication networks. *PADS 2002*: 107-114.

[27] S. Franklin and A. Graesser. (1996). Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag.

[28] R. Fujimoto. (1999). Exploiting temporal uncertainty in parallel and distributed simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp 46-53. IEEE Computer Society, May 1999.

[29] R. Fujimoto. (2000). Parallel and Distributed Simulation Systems. Wiley Interscience, 2000.

[30] L. Gasser and K. Kakugawa. (2002). MACE3J: Fast Flexible Distributed Simulation of Large-Grain Multi-Agent Systems. *AAMAS 2002*, 745-752.

[31] M. P. Georgeff and A. Lansky. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp 677-682, Seattle,WA, N. Gilbert.

[32] B. P. Gerkey, R. T. Vaughan, and A. Howard. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, pp 317-323.

[33] B. Groselj. (1995). CPSim: A Tool for Creating Scalable Discrete-Event Simulations. *Proceedings of 1995 Winter Simulation Conference*, pp. 579-583.

[34] A. Helsinger, M. Thome and T. Wright. (2004). Cougaar: a scalable, distributed multi-agent architecture. *SMC (2) 2004*, 1910-1917.

[35] J. H. Holland. (1995). Hidden Order: How Adaptation Builds Complexity. Addison-Wesley, Reading.

[36] M. Hybinette and R. Fujimoto: Cloning parallel simulations. (2001) *ACM Trans. Model. Comput. Simul. 11(4)*, pp. 378-407.

[37] M. Hybinette, E. Kraemer, Y. Xiong, G. Matthews and J. Ahmed. (2006). SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure. *Winter Simulation Conference 2006*: 926-933.

[38] H. Iba. (1999). Evolving multiple agents by genetic programming. In Spector, L., Langdon,W. B., O'Reilly, U. M., and Angeline, P. J., ed. Advances in Genetic Programming 3. 447-466.MIT Press, Cambridge, MA, USA.

[39] D. R. Jefferson and H. Sowizral.(1982). Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control. Technical Report N-1906-AF, RAND Corporation, December 1982.

[40] D. R. Jefferson. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404-425, July.

[41] S. F. Kaplan, L. A. McGeoch and M. F. Cole. (2002). Adaptive caching for demand prepaging. *ISMM 2002: Proceedings of the 3rd international symposium on Memory management.*

[42] D. Kinny. (1990). Measuring the Effectiveness of situated agents. Technical Report 11, Australian AI Institute, Carlton, Australia.

[43] D. Kinny, M. Ljungberg, A. S. Rao, L. Sonenberg, G. Tidhar and E. Werner.(1992). Planned Team Activity. *MAAMAW 1992*: 227-256.

[44] M. Lees. (2002). A History of the Tileworld Testbed. Computer Science Technical Report No. NOTTCS-WP-2002-1 University of Nottingham.

[45] M. Lees, B. Logan, R. Minson, T. Oguara and G. Theodoropoulos. (2004). Modelling environments for distributed simulation. *Environments for Multi-Agent Systems: Proceedings of the the 1st International Workshop(E4MAS'04)*, D. Weyns, H. V. D. Parunak, and F. Michel, Eds. Number 3374 in *LNAI. Springer*, pp. 150-167.

[46] M. Lees, B. Logan and G. Theodoropoulos. (2006). Agents, Games and HLA. *Simulation Modeling Practice and Theory.*

[47] M. Lees, B. Logan and G. K. Theodoropoulos. (2007). Distributed simulation of agent-based systems with HLA. *ACM Trans. Model. Computer Simulation*, 17(3).

[48] Y. A. Liu and T. Teitelbaum.(1995). Caching Intermediate Results for Program Improvement. *PEPM 1995*: 190-201.

[49] Y. H. Low et.al. 1999. Survey of Languages and Runtime Libraries for Parallel Discrete Event Simulation. http://www.ntu.edu.sg/home/yhlow/papers/low99survey.pdf.

[50] M. C. Lowry, P. J. Ashenden and K. A. Hawick. 2000. A Testbed System For Time Warp in Java. http://www.dhpc.adelaide.edu.au/reports/087/dhpc-087.pdf

[51] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan and G. Balan. (2005). MASON: A Multiagent Simulation Environment. *SIMULATION*, 2005, 517-527

[52] C. M. Macal and M. J. North. (2005). Tutorial on Agent-Based Modelling and Simulation. *Proceedings of the 2005 Winter Simulation Conference.*

[53] D. E. Martin, P. A. Wilsey, R. J. Hoekstra, E. R. Keiter, S. A. Hutchinson, T. V. Russo and L. J. Waters. (2003). Redesigning the WARPED Simulation Kernel for Analysis and Application Development. *ANSS '03: Proceedings of the 36th annual symposium on Simulation.*

[54] E. Mascarenhas, F. Knop and V. Rego. (1995). ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads. *Winter Simulation Conference 1995*, 690-697.

[55] P. V. Mockapetris.(1987), November. RFC 1034: Domain names, concepts and facilities. Obsoletes RFC0973, RFC0882, RFC0883. See also STD0013 Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC2065, RFC2181, RFC2308. Status: STANDARD.

[56] N. Minar, R. Burkhart, C. Langton and M. Askenazi. (1996). The Swarm Simulation System: Toolkit for Building Multi-agent Simulations. Technical report. http://www.cse.nd.edu/courses/cse598j/www/Resources/overview.pdf

[57] D. Nicol, M. Johnson, A. Yoshimura and M. E. Goldsby. (1998). IDES: A Java-based Distributed Simulation Engine. 1998 International Workshop on Modeling Analysis and Simulation of Computer and Telecommunication Systems. Montreal, Canada, pp. 233-240.

[58] C. Nikolai and G. Madey. (2009). Tools of the Trade: A Survey of Various Agent Based Modeling Platforms. *Journal of Artificial Societies and Social Simulation*, vol. 12.

[59] L. Nolle, P. K. C. Wong and A. A. Hopgood. (2001). DARBS: A Distributed Blackboard System. *Research and Development in Intelligent Systems XVIII*, Bramer, Coenen and Preece (eds.), Springer, pp 161-170.

[60] M. J. North, N. T. Collier and J. R. Vos. (2006). Experiences in Creating Three Implementations of the Repast Agent Modeling Toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1-25, January.

[61] S. Park and L. Leemis. (2001). Discrete-Event Simulation: A First Course. College of William and Mary. http://www.cs.wm.edu/ esmirni/Teaching/cs526/.

[62] D. C. Parker, S. M. Manson, M. A. Janssen, M. J. Hoffmann, and P. Deadman. (2003). Multi-Agent Systems for the Simulation of Land-Use and Land-Cover Change: A Review. *Annals of the Association of American Geographers*, 93(2): 314-337.

[63] M. E. Pollack and M. Ringuette.(1990). Introducing the Tileworld: Experimentally Evaluating Agent Architectures. *AAAI 1990*: 183-189.

[64] M. E. Pollack, D. Joslin, A. Nunes, S. Ur and E. Ephrati. (1994). Experimental investigation of an agent commitment strategy. Technical Report 94:31, Pittsburgh, PA 15260.

[65] B. R. Preiss. (1990). Yaddes - Yet Another Distributed Discrete Event Simulator: User Manual. Tech. Report, Department of Electrical and Computer Engineering, University of Waterloo, Canada.

[66] B. R. Priess and and C. K. W. Wan.(1999) The Parsimony Project: A Distributed Simulation Testbed in Java. In *Proc. 1999 International Conference On Web-Based Modelling and Simulation*, volume 31 of Simulation Series, pp 89-94, San Francisco, CA, January 1999.

[67] W. Pugh and T. Teitelbaum.(1989). Incremental Computation via Function Caching. *POPL 1989*: 315-328.

[68] P. L. Reiher. (1990). Parallel Simulation Using the Time Warp Operating System. *Proceedings of 1990 Winter Simulation Conference.*

[69] P. L. Reiher, F. Wieland, D. R. Jefferson. (1989). Limitation of optimism in the time warp operating system. *Winter Simulation Conference 1989*, 765-770.

[70] P. Riley and G. Riley. (2003). SPADES – A distributed agent simulation environment with software-in-the-loop execution. In *Proceedings of the 2003 Winter Simulation Conference (WSC-2003)*, 817-825.

[71] R. Ronngren, M. Liljenstam, R. Ayani and J. Montagnat. (1996). A Comparative Study of State Saving Mechanism for Time Warp Synchronized Parallel Discrete Event Simulation. *IEEE Proceedings of Simulation.*

[72] S. J. Russell and P. Norvig. (2003). Artificial Intelligence: A Modern Approach, Prentice Hall, USA.

[73] B. Samedi. (1987). A distributed algorithm to detect a global state of a distributed simulation system. IFIP Conference on Distributed Processing, October 1987.

[74] B. Schattenberg and A. M. Uhrmacher. (2001). Planning agents in JAMES. *Proceedings of the IEEE 89, 2 (Feb.)*, pp. 158-173.

[75] A. Sloman and R. Poli. (1999). Sim agent: A toolkit for exploring agent designs, in M. Wooldridge, J. Mueller and M. Tambe, eds, *Intelligent Agents Vol II (ATAL-95)*, Springer-Verlag, pp. 392–407.

[76] J. S. Steinman. (1992). SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal in Computer Simulation*, Vol.2, No.3, pp. 251-286.

[77] J. S. Steinman and Jennifer W. Wong. (2003). The SPEEDES Persistence Framework and the Standard Simulation Architecture. *PADS 2003: Proceedings of the seventeenth workshop on Parallel and distributed simulation.*

[78] J. S. Steinman. (2005). The Warp IV Simulation Kernel. *PADS 2005: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation.*

[79] S. Strogatz.(2007). The End of Insight. In Brockman, John. What is your dangerous idea? Harper Collins, 2007.

[80] (2006) Swarm: A Platform for Agent-Based Models. http://www.swarm.org/

[81] Y. M. Teo and Y. K. Ng. (2002). SPaDES/Java: Object-Oriented Parallel Discrete-Event Simulation. *Proceedings of the 35th Annual Simulation Symposium 2002*, 245-252.

[82] P. M. Torrens. (2004). Simulating Sprawl: A Dynamic Entity-Based Approach to Modelling North American Suburban Sprawl Using Cellular Automata and Multi-Agent Systems, Ph.D. Thesis, University College London, London.

[83] A. M. Uhrmacher. (1997). Concepts of Object-and Agent-Oriented Simulation. *Transactions of the Society for Computer Simulation International.* Vol. 14, No. 2, pp. 59-67.

[84] A. M. Uhrmacher and K. Gugler. (2000). Distributed, parallel simulation of multiple, deliberative agents. *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS-2000)*, 101-108.

[85] A. M. Uhrmacher. (2001). Dynamic structures in modeling and simulation: A reflective approach. *ACM Trans. Model. Comput. Simul. 11, 2*, pp. 206-232.

[86] G. Vulov, T. H. He and M. Hybinette. (2008). Quantitative assessment of an agent-based simulation on a time warp executive. *WSC '08: Proceedings of the 40th Conference on Winter Simulation.*

[87] J. Waldorf and R. Bagrodia. (1994). MOOSE: A Concurrent Object-Oriented Language for Simulation. *International Journal of Computer Simulation*, Vol.4, No.2, pp. 235-257.

[88] K. Walsh II and E. G. Sirer.(2004). Simulation of large scale networks I: staged simulation for improving scale and performance of wireless network simulations. *Winter Simulation Conference 2003*: 667-675.

[89] J. West and A. Mullarney. (1988). ModSim: A Language for Distributed Simulation. *Proceedings of SCS Multi-conference on Distributed Simulation*, pp. 155-159.

[90] P. Wonnacott and D. Bruce. (1995). The Design of APOSTLE - A High-Level, Object-Oriented Language for Parallel and Distributed Discrete Event Simulation. *Proceedings of 1995 Winter Simulation Conference.*

[91] M. Wooldridge and N. R. Jennings.(1994). Formalizing the cooperative problem solving process. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence (IWDAI-94)*, pp 403-417, Lake Quinalt, WA, July 1994.

[92] M. Wooldridge and N. R. Jennings. (1995). Agent Theories, Architectures, and Languages: a Survey. Wooldridge and Jennings Eds., Intelligent Agents, Berlin: Springer-Verlag, 1-22. 1995.

[93] Y. Xiong, M. Hybinette and E. Kraemer. (2008). Transparent and adaptive computation-block caching for agent-based simulation on a PDES core. *WSC 2008: Proceedings of the 40th Conference on Winter Simulation*.

[94] B. P. Zeigler and S. Vahie. (1993). DEVS formalism and methodology: unity of conception/diversity of application. *Winter Simulation Conference 1993*, pp. 573-579.