MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems

Les Gasser Kelvin Kakugawa
Graduate School of Library and Information Science
University of Illinois at Urbana-Champaign
Champaign, IL 61820, USA
{gasser, kakugawa}@uiuc.edu

ABSTRACT

Scientific study of multi-agent systems (MAS) requires infrastructure such as development testbeds and simulation tools for repeatable, controlled experiments with MAS structure and behavior. Testbeds and simulation tools are also critical for MAS education and development. A number of MAS testbeds currently exist, but to date none meets in a comprehensive way criteria laid out by many analysts for general, scientific, experimental study of MAS by a large community. Moreover, none really scales to very large MAS or exploits the power of modern distributed computing environments such as large multiprocessor clusters and computational grids. Because of this, and specifically to fulfill widespread need for tools supporting distributed collaborative scientific research in large-scale, large-grain MAS, we created the MACE3J system, a successor to the pioneering MACE testbed.

MACE3J is a Java-based MAS simulation, integration, and development testbed, with a supporting library of components, examples, and documentation, distributed freely. MACE3J currently runs on single- and multiprocessor workstations, and in large multiprocessor cluster environments. The MACE3J design is multi-grain, but gives special attention to simulating very large communities of large-grain agents. It exhibits a significant degree of scalability, and has been effectively used in fast simulations of over 5,000 agents, 10,000 tasks, and 10M messages, and on multiprocessor configurations of up to 48 processors, with a future target of at least 1000 processors.

This paper presents MACE3J design criteria and our approach to a number of critical tradeoffs that, to our knowledge, have not previously been treated explicitly in MAS literature or platforms. We present the innovative features of the MACE3J architecture that contribute to its breadth, flexibility and scalability, and finally give results from the use of MACE3J in real experiments in realistic MAS domains, both simple and complex.

AAMAS'02, July 15-19, 2002, Bologna, Italy.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems, languages and structures; I.6.7 [**Simulation and Modeling**]: simulation support systems, types of simulation—*parallel, distributed, discrete-event*; C.1.4 [**Parallel Architectures**]: distributed architectures; C.4 [**Performance of Systems**]: modeling techniques

1. INTRODUCTION

Scientific study of multi-agent systems (MAS) requires infrastructure such as development testbeds and simulation tools to perform repeatable, controlled experiments with MAS structure and behavior. Testbeds and simulation tools are also useful, if not critical, for MAS education and development [9]. MAS researchers do use simulation and system construction tools widely, and a number of MAS testbeds have been built (e.g., [2, 6, 11, 22]). Unfortunately, to date none of these testbeds meets in a comprehensive way the criteria laid out by a number of analysts for general, scientific, experimental study of MAS by a large community (e.g., [5, 9, 12, 15, 18]). Because of this, and specifically to fulfill our needs for a widely-available tool for scientific research in large-scale, large-grain MAS, we launched the construction of the MACE3J system, a successor to the original MACE testbed [6].

MACE3J is a Java-based MAS simulation, integration, and development testbed, with a supporting library of components, examples, and documentation, distributed freely. MACE3J currently runs on single- and multiprocessor workstations, and in multiprocessor cluster environments such as Sun multiprocessor clusters or the SGI Origin (both of which we use). The MACE3J design is multi-grain, but gives special attention to simulating very large communities of large-grain agents. It exhibits a significant degree of scalability, and has been effectively used in fast simulations of over 5,000 agents, 10,000 tasks, and 10M messages, and on multiprocessor configurations of up to 48 processors, with a future target of at least 1000 processors.

This paper presents MACE3J design criteria and our approach to a number of critical tradeoffs that, to our knowledge, have not previously been treated explicitly in MAS literature or platforms. We present the innovative features of the MACE3J architecture that contribute to its breadth, flexibility and scalability, and finally give results from the use of MACE3J in real experiments in realistic MAS domains, both simple and complex.

2. RELATED WORK

Simulation and experimentation is a mainstay of MAS research, but there has been surprisingly little analysis of the general requirements for a range of MAS needs. This general analysis is important to identify the shared requirements of many projects and how to address them with new infrastructures targeted at the community level, rather than at individual projects. Based on such analyses, community level infrastructures will help strengthen the scientific quality of MAS research by improving the sharing of models, data, and results. Two such recent analyses are presented in [5, 9]; an earlier one is [12]. Here we review several projects related to the design goals and approaches of MACE3J, with the aims of illustrating a) the breadth of need, b) the specialized focus of many tools, c) the range of functionality, flexibility, and application considered important, and d) specific approaches taken. There is a number of MAS testbed and simulation projects, and space precludes treatment of all of them, so we present a representative cross-section and summary of the main themes.

2.1 MACE

The original MACE [6] was a truly-distributed object-oriented system running on early-generation workstation networks and first-generation distributed- and shared-memory multiprocessors. The 1986 version of MACE included implementations of the following ideas that influenced several other MAS testbeds, and have now become mainstream: *i*) user-interface agents; *ii*) *acquaintance* databases that model attributes of other agents; *iii*) multi-level "agentified" infrastructure (experiment-construction agents and a collection of system agents in the MACE kernel); *iv*) 'pluggable' testbed interfaces that allow seamless agent transitions from simulated contexts to actual real-world operating conditions; *v*) visualization and display of multi-agent system behavior as a central issue and research topic (that is still under-studied; cf. [16]); and *vi*) the philosophy of using a specific social model to design a multi-agent system. However, MACE included no notion of environment models (cf. [2, 4, 15, 22]

2.2 High-Level Architecture

HLA/RTI [14] is a very generic, language-independent *architectural specification* (not implementation) for an infrastructure that integrates simulation components ("federates") into robust, controllable aggregates ("federations"). The "RunTime Interface" (RTI) is a specification of how to implement the core services of HLA, e.g., with specific Java or C++ bindings. An RTI implementation provides interfaces and services that federation designers can call upon to integrate active simulations and synchronize them, including registration, time management, inter-federate interaction, definitions of shared objects. HLA/RTI is not specifically agent-based, and typical agent-oriented services (message passing, agent communication languages, middleware services, addressing, coordination infrastructures, etc.) are not an intrinsic part of HLA/RTI. From the perspective of the MACE3J project, it is most appropriate to view HLA/RTI as source of language, metaphors and specifications for MACE3J, a potential integration substrate beneath MACE3J, and/or a potential integration target for standalone MACE3J simulations (i.e., one MACE3J simulation as single HLA/RTI federate). There is no free, open-source implementation of HLA/RTI. Though quite complete as a general integration approach, model construction overhead is high because many kinds of interaction must be fully specified [19, 20].

2.3 SSF

The Scalable Simulation Framework (SSF) [3] is, like HLA, a *specification* for a set of entities and simulation services. Again, it isn't specifically agent-based, and typical agent-oriented services are not an intrinsic part. There is a C++ implementation of SSF and apparently some work on a Java implementation. SSF was originally designed for very large discrete event simulations of very large scale distributed processes such as operating layers of the internet, and the C++ SSF implementations have been used in 1M node experiments. The SSF is a very general model with just five core entities: `Event`, `Entity`, `inChannel`, `outChannel`, and `process`. SSF introduces the notion of flexible *coalignment* among a group of entities to a common timeline. The coalignment concept is a very clean abstraction for simulation, and has been adopted in MACE3J. Like HLA, it is most appropriate to view SSF as source of language, metaphors and specifications for MACE3J.

2.4 PDES-MAS

PDES-MAS [15] is a relatively new project, whose main focus is how to use optimistic (look-ahead) simulation techniques to improve the performance of environment-based MAS models. When a collection of agents interacts with a globally shared environment, the environment can create a speed bottleneck even if the individual agents are concurrently simulated. The approach of PDES-MAS is to dynamically partition a shared environment into "interest areas" which provide local constraints on agent-agent interaction. Interaction within separate interest areas can be executed concurrently, helping gain overall speedup of a simulation. To our knowledge, there is no current free software implementation available.

2.5 MASS

The UMASS Multi-Agent System Simulator (MASS) is designed for creating sophisticated autonomous agents that are reactive to their environment, and that perform goal-oriented decision-making under constraining conditions such as deadlines and resource tradeoffs [22]. MASS agents are created using the component-based *Java Agent Framework* (JAF), and environment simulation uses the Task Analysis, Environmental Modeling and Simulation language TAEMS [4]. The environment itself contains global shared variables that are used to control agent coalignment during a simulation. MASS also has exploited the notion of making seamless, flexible transitions between simulated and real-world tasks. Using JAF, the MASS-specific components in an agent can be replaced with components that interact directly with the real world, allowing for a separation between the agent's logic and its environmental interactions. The agent and environment models in MASS are specific to the JAF and TAEMS models.

2.6 RePast and Swarm

RePast [2] is a derivative of the ideas in the well-known Swarm toolkit, developed in Java (which gives it a potentially wider audience than Swarm in Objective C). RePast contains flexible agent and environment modeling facilities,

display tools that include graphs and movies, and flexible data capture, streaming and monitoring facilities. A particular distinguishing characteristic of RePast is it nested timing model, in which allows a designer to model a very complex hierarchy of simulation events. RePast contains no communication (messaging) facility, and is designed for single-processor environments; the timing model is particularly tricky to distribute across multiple processors.

2.7 MADKit

MADKIT [11] is an agent construction testbed in Java characterized by a novel kernel architecture. One of MADKIT's aims, like the original MACE, is to integrate a specific social model as a foundation for system design. MADKit is based on kernel-level support for systems built of *groups* of agents, each of which holds one or more *role*s. Groups and roles provide structuring abstractions that pervade all of MADkit's facilities. Like the original MACE, MADKit uses system-level agents to build the framework for user-level multi-agent interactions. One advantage of this approach is that the framework itself can change and adapt as the modeling and simulation needs change. Every MADKit agent an interface by default, and MADKit contains a heterogeneous, flexible set of tools for building agents including Java, JPython, JScheme languages

3. MACE3J DESIGN GOALS AND TRADEOFFS

These specifications, frameworks, and testbeds illustrate the range of issues with which MAS infrastructure researchers have grappled. Taken together, they provide an exciting potential landscape for experimentation. However there are many gaps, summarized here, and validated by our own experience with most of the actual software. First, none of the agent-specific testbeds have facilities for importing agents designed for other testbed environments. In general, their specificity militates against agent-technology sharing. As an integration framework HLA comes closest, but it provides no specific enactment facilities of its own, and HLA is just a specification. Second, few testbeds are freely available, highly operational, well-documented, scalable, *and* oriented to large-grain MAS. RePast, very stable and adequately documented, probably comes closest, but it contains no communication model and isn't scalable to multiprocessors. Finally information-gathering is the aim of experimentation, and few of these testbeds have rich facilities for instrumentation, information gathering, and display (Again, RePast comes closest, to our knowledge). With this background in mind, we describe MACE3J's design goals and tradeoffs.

The overriding aim of MACE3J is to support scientific approaches to the study of Multi-Agent Systems, in a way that also supports practical application. In our view, any scientifically oriented simulation testbed should satisfy at least the following three objectives:

1. Transitionable Models: A simulation is a model of a process or artifact–it's not the real thing. Yet it is often desirable to "unplug" simulated objects and transition them in to real environments for a simulation-to-use development trajectory or simply for dual-use. But simulation *per se* has different aims than use, such as explorations of the efficacy of conceptual simplifications entailed by modeling, or repeatable testing under specific controlled scenarios. The easier it is to migrate an agent into a distributed computing environments, the more positive exposure and application agent technology is likely to have.

2. Generation of knowledge about behavior and structure: Models are built because researchers want simplified ways to study phenomena or objects. Hence a central aim is the generation of knowledge, and a simulation environment should support the gathering, representation, and analysis of information in order to do so. A number of so-called simulations and simulation testbeds overlook this key aspect.

3. Repeatability and control: Many widely used experimental methods require controlling conditions and behavior, and simulations are very good at this, within assumptions and limitations of model simplification and validation. There are typically two approaches. In *Monte-Carlo simulations*, experimenters "control for" extrinsic fluctuations in interactions and behavior by allowing them to occur. By making many runs per scenario, statistical manipulations (e.g., averaging) can be used to eliminate the effects of irrelevant fluctuations. An alternative approach to eliminating the effects of fluctuations in experiments is to make fewer runs per scenario, but to randomize aspects (like execution order or message delivery sequencing) that could introduce unwanted experimental artifacts. The randomization approach can make the experimental process more efficient because fewer runs are needed, but experimenters need to give careful attention to what aspects will be controlled by randomization. To support the randomization approach, a testbed needs to support careful, selective control of deterministic action. Ideally, repeatability and control should be policy choices, and simulation support should allow flexibility on them.

Toward these ends, MACE3J has extensive support for:

1. A selectable combination of deterministic (simulation-driven), user-driven, environment-driven, and/or probabilistic control of simulation events. This allows simulations to be re-run exactly, while supporting probabilistic control of behavioral and timing aspects of simulations such as message delay and system failure (e.g., failure of message delivery or of execution).

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).

4. Reusable components for constructing ActiveObjects[1], environments, and experiments, coupled with the ability to flexibly import these components from other projects.

[1]MACE3J generalizes the concept of "agents" to *ActiveObjects*, which are defined in MACE3J with a set of interfaces. The MACE3J concept of ActiveObjects captures core functionality that allows for implementation of many different types of "agents", so we use the term in this paper to denote the foundations for a range of typical agent types.

3.1 MACE as a General Model of Collectives and Interactions

In addition to these straightforward, instrumental aims, from a more general and more philosophical point of view MACE is also an experiment in new ways of conceptualizing interaction and collective behavior among threaded ("active") objects in general. In effect, interactions among MACE3J's ActiveObjects (or agents) may be seen as a specialization of the problem of interaction among objects in collections. Ideally, an agent-based simulation environment should be founded on a general and flexible model of object organization. Toward this end, a collection of primitive organization and interaction concepts can provide the framework upon which to construct a flexible simulation environment like MACE3J. (This idea is in harmony with, for example, the approach taken in MadKit [11]. MADKit's underlying philosophy of using organization-structuring concepts as a foundation for agent-support environments is in harmony with our approach.

Simulations built with MACE are designed to be collections of threaded distributed objects [6, 7], and the structure of the MACE system is an attempt to build a very general conceptual framework for dynamically specifiable control of systematic interactions between such objects. Our idea is to exploit a very general interaction model to construct a specific set of constrained object interactions that can be useful as an agent-based simulation, while maintaining the flexibility to change the fundamental nature of the simulation model fairly easily. On the one hand, we could have created a monolithic software system with a single architecture of control, as has been done in every system reviewed above. On the other hand, we could have constructed an open, fully flexible actor-based system wherein all control within MACE and among simulated objects alike was opportunistic and message driven. How should we maintain flexibility and yet have an efficient, targeted framework?

To answer this, we have had to consider carefully the role and meaning of message-based interaction in a distributed object system, as differentiated from procedure call interactions or co-routines. In our view, the two aspects that most clearly differentiate these two positions are a) message-oriented interactions have a character of *mutual control choice* that procedure calls do not, and b) message-oriented interactions imply that messages require (i.e. are open to) semantic interpretation by message receivers.

For message senders, mutuality of control means that the choice to send a message is the result of some explicit, situationally-grounded computation of the need for a message and of its contents. This computation is a control computation that is carried out locally by the sender. For message receivers, the mutuality means that a receiver can choose whether, when, and how to react to a message. This entails control computations in the receiver that are contexted against the receiver's local situation. "Mutual selection" as a basis for distributed control was first introduced by Davis and Smith in the Contract Net system, and specifically oriented toward market-like distributed problem solving. Here we are talking more generally about the natural implications of any asynchronous message-based communications and control among semi-autonomous objects that actually interpret the messages.

Another set of choices that objects or agents make in message-based communications is a) what to send in a message in order to foster an expected response, and b) how to act on a received message. This is the general problem of how to associate a communicated form with a meaning or (expected) response, i.e. the general problem of language. (This particular structural-functional view of language as the problem of consistently associating linguistic forms with localized meanings is widely used in the Language Evolution community; see e.g., [1, 21]. Both of these choices (what to send and how to interpret what is received) are made as a result of some computation made by the sender or receiver object. Such dynamic interpretation clearly adds to the flexibility of a system, but it also incurs additional computational overhead. In any case, the explicitly interpretive computation is what differentiates message-oriented communication from procedure calls, which entail no explicit interpretation step, just reactive action. For MACE, the key issue has been which internal, external, and simulated interactions should incorporate this interpretive layer, and which should be pure, highly-structured, procedure calls.

3.2 Design Dilemmas and Tradeoffs

3.2.1 Agent-Centered vs. Testbed-Centered Autonomy

Agents are typically defined as objects with autonomy of knowledge, control (decisionmaking), and interaction. However, the degree of control needed for deterministic simulation interacts with the autonomy of an ActiveObject in context. This is a fundamental issue for agent-based simulation and it appears in the context of time management and environmental interaction. In an *autonomous-objects model*, ActiveObjects are always running in their own processes, and they explicitly request activation and synchronization services from the simulator kernel. Under this model, there may be several general classes of synchronization (possibly in a nested synchronization model, ala RePast [2]), and user-defined synchronization classes for events. ActiveObjects would 'subscribe' to one or more of these classes of events and be synchronized on them. Also from this perspective, the simulator "traps" some object actions and executes them under a controlled simulation model. Other object actions are not trapped and proceed in a 'natural' way. So, for example, a communication action may be trapped and handled by the MACE3J MessagingSystem (MMS), or it could be sent via some sort of 'real' communication service external to MACE3J. In this way the MMS becomes part of the virtual environment supplied by the testbed, and whether it gets used or not is be a matter for specification of an experiment.

An alternative is an *autonomous simulator model* in which the simulator kernel creates a thread for an ActiveObject and starts the object in that thread only when needed (In many cases, not all potentially-active objects are actually active during each time slice). Under this model, ActiveObjects have no control over synchronization, are not fully autonomous, and have environmental interactions that are strongly mediated by the testbed. The choice between these two models has strong interactions with the base ActiveObject model of the testbed.

There is a fairly simple way of merging these two approaches, which combines object requests with a proxy-oriented interface between the ActiveObject and the testbed, and this is MACE3J's approach. ActiveObjects can request

either or both of two kinds of services from the simulator (specifically from an `ActivationGroup`: threads and/or synchronization services. Requesting a thread supplies an active object with a thread of control, and the object is responsible for relinquishing control and passing it back to MACE3J when appropriate (this is also the approach taken in HLA and SSF).

Requesting a "pulse" (here using the terms of the MASS simulator [22]), would yield a control token sent to the object at the appropriate time, knowing that the object was already active (had a process thread) and needed a synchronization token to begin processing for that time unit. HLA also provides this method.

3.2.2 Determinism and Control vs. Opportunism and Realistic Dynamics

Testbed *environments* for agents can be modeled and simulated or they can be provided via interfaces to some external reality. There are tradeoffs here because the external reality, while possibly patterned, does not necessarily exhibit the degree of determinism that can lead to repeatable, controllable experiments. The deep issue for experimentation and research design is to understand the ramifications of the loss of realism when an environment is completely simulated, and to trade these off against the need for repeatability and control. This issue has also been raised in [12, 22, 4]. Again the answer seems to be to provide a variety of mechanisms to implement a range of policies. Thus, extending the approach in [22], MACE3J provides flexible and selective transitions between actual services and simulated ones without the need for agent reconfiguration, by using proxies to represent both ActiveObject-to-testbed interactions and testbed-to-environment interactions. In this way, both the ActiveObjects and the testbed become isolated and encapsulated with standard interfaces.

This issue also impacts the related issue of whether and how to migrate ActiveObjects seamlessly from a testbed environment to real environments. Assuming that the internal logic of an agent or ActiveObject is consistent with real-world operation, the proxy approach of MACE makes flexible transition effective.

3.2.3 Kernel Centralization or Distribution

If testbeds are to be scalable to large agent populations, they must be designed for multiprocessing environments. The most prevalent multiprocessor architectures use distributed memory, and this presents specific challenges for agent testbeds. Of particular interest is the experimental overhead associated with programming, starting and managing a collection of resources that are both distributed and concurrent. The first MACE testbed [6] was designed for distributed memory environments and used a collection of distributed kernels. This meant that programmers had to manage loads, experiment startups, and more. (This is one reason we were led to and "agentified" testbed system in the earlier MACE, so that agents could manage large parts of this burden—necessity being the mother of invention.) Modern environments for multicomputing however, are beginning to exploit the technology of *Single System Image* (SSI) clusters, and the Java Virtual Machine (JVM) is no exception. (See [13] for pointers to this literature, including the cJVM, Jessica, and Hyperion projects.) Since MACE3J's design is targeted for large-grain ActiveObjects, and running many of them within the same JVM can build very large JVM images very quickly. This has both processing locality effects and very strong performance effects when real memory limits are reached. Maintaining a SSI testbed with the JVM is much easier because the JVM and the underlying cluster software manage process mapping, load balancing, inter-processor communication, thread startup, etc. Also, startup of an SSI image is likely to be more reliable due to easily centralized control over interactions. However, multiple JVMs have the advantage of being individually smaller and potentially available on a wider array of machines. At present, MACE3J relies upon the SSI approach, in part because we are interested to explore the limits of large-grain SSI clustering for MAS.

3.2.4 Agent Granularity

There is an obvious tradeoff between the number of ActiveObjects that can be run realistically with a given set of resources, and the grain size of the ActiveObjects. This tradeoff also manifests itself in a number of less-obvious ways, including the control behavior of synchronization and speed optimization. An testbed that provides its own round-robin activation loop for agents (such as RePast) can minimize context-switching overhead, but can never scale to large numbers of large-grain agents. The process creation and management overhead of a distributed concurrent environment is only worthwhile when the benefits of concurrency outweigh the costs of this overhead, and this is only the case when the overhead is small relative to the execution time of the ActiveObjects. This is exactly the case for large-grain ActiveObjects. Nonetheless to address this tradeoff in a variety of environments for a variety of ActiveObject types, the MACE3J's architecture implements several types of enactment ranging from a (randomized) round robin processing loop in a single timesliced thread (for large collections of smaller agents running on a uniprocessor), to individual concurrent processes in separate threads (for large grain agents in a SSI/multiprocessing environment).

4. ARCHITECTURE

To address these tradeoffs, we have designed a flexible, lightweight architecture that we view as an extension to the Java Virtual Machine.

MACE3J has two central organizing ideas—one concerning the nature of collectives whose members act in relation to one another, and the other concerning the nature of individual members of those collectives. The foundation for collectives in MACE3J is a concept (and implemented class) called the `ActivationGroup`. The MACE3J concept for representing a members of such a collective is "ActiveObject", which is not a class, but instead is a family of implemented interfaces[2] that define interactions with batch of services with progressively greater sophistication.

4.1 ActivationGroups and their Services

[2]This subsection assumes familiarity with the Java `interface` concept. Basically any class that *implements an interface* provides the set of methods with specific argument types that the interface specifies. By doing this, the class's type also becomes that of the implemented interface, and other objects know how to call it in conformance with the interface definition.

In MACE3J a defined set of ActiveObjects needs access to some commonly organized services. These services should be organized in common because of a) the nature of coordination relationships among objects that are *ActivationGroup* members and b) the desire for certain types of control over interactions among ActiveObjects. The common MACE3J services for an *ActivationGroup* are implemented as objects, as follows:

`Registrar` is an object that manages membership in the ActivationGroup and provides the proxies for ActiveObjects to other MACE3J services.

`Scheduler` is an object that decides what ActivationGroup member objects to enact in what order. The `Scheduler` accepts *enactment requests* and may randomize object execution, object failure, etc..

`TimeManager` is an object that decides when and how to advance a clock that is global to the ActivationGroup.

`MessageSystem` is an object that provides message transfer services for registered ActiveObjects. The `MessageSystem` may randomize message delivery order, message failure, noise in message content, etc.

`Enactor` is an object that provides enactment services, that is, simulated invocations, MACE3J-allocated threads, or signals to act ("pulses" in [22]) for objects that have their own autonomous threads.

`Environment` is an object that serves as a gateway to a user-defined MACE3J environment model.

`ControlStructure` is an object that describes the flow of control among MACE3J services at a metalevel. For example, the standard MACE3J `ControlStructure` deterministically implements a *conservative synchronization* regime using a fork-join concurrency model. The `TimeManager` only advances when it has determined that no ActiveObject will receive any events (e.g. messages) from the past. A more flexible `ControlStructure` might "agentify" MACE3J services and interleave their execution in a message-driven manner.

`Randomizer` is an object that provides (pseudo) random numbers, probability distributions, and randomization services based on them.

To support MACE3J's aim of flexible transfer between simulated and real environments, agents interact with many of these services through registered proxies, and the services interact with their own "real" back-end services with proxies as well. Thus, the Enactor object never knows whether the agent it is enacting actually has its own autonomous process thread, possibly on another machine, or is being run under a MACE3J thread, and never knows whether its startup method takes place through a procedure call or via a message sent to a remote location; the proxy hides this. Agents, too, may not know whether the services such as `Enactor`, `MessageSystem`, or `Environment` are being provided by MACE3J itself through simulation, or are passed through to a"real" back-end service. In this way, MACE3J's design shifts the burden of transparency to the system itself, instead of requiring a change in agent code to shift to the "real world" as a substrate (as in JAF [22]). As long as MACE3J services use the proxies of ActiveObjects and back-end services, it doesn't matter if the services are real or part of a simulation.

4.2 Defining Agents

A critical design goal of MACE3J is to support a very wide variety of user-built agent types without undue programming or code reconfiguration overhead. Unlike many other testbeds, (but like the original MACE system) the MACE3J kernel makes virtually no commitment to the internal architecture of the ActiveObjects that run atop it. The root interface of the ActiveObject family is the the `Registerable` interface. Any Java object can implement `Registerable` and then register with an `ActivationGroup` instance of MACE3J. A `Registerable` ActiveObject need not have a thread of control, and may be executed via a call from some other Java object. If it desires to receive enactment services from MACE3J, it implements an extension to `Registerable` called the `Enactable` interface and makes at least one *enactment request* of the `ActivationGroup` with which it is registered. ActiveObjects running under MACE3J need not be time-synchronized (or *time-constrained* to use the HLA term [14]), but they may avail themselves of their `ActivationGroup`'s time management services by implementing an interface extension to `Enactable` called `TimeSynchronized`. Next, MACE3J ActiveObjects need not communicate via messages, but they may do so by implementing an interface extension to `Registerable` called `Messageable`, which gets them access to their `ActivationGroup`'s `MessageSystem`. Additional interfaces in the ActiveObject family include: `Sensate`, describing agents with "perceptual" access to a MACE3J environmental model; `Effectual`, describing agents that can actuate environmental effectors; `Viewable`, describing agents that output and present information through a variety of media; and `Controllable`, describing agents that can be controlled by other agents or users. Typically, skeleton implementations of these interfaces are also given as *abstract classes* that a user can easily extend. Using elements of this family of interfaces, a wide variety of ActiveObject types can be implemented using MACE3J services. For example, it is easy to define an ActiveObject that sends messages using its `ActivationGroup`'s `MessageSystem`, but is neither `TimeSynchronized` nor `Enactable` by MACE3J, running under its own control thread. Or, one can create a completely simulated ActiveObject that sends no messages (ala [2]).

For simplicity, MACE3J defines extended interfaces called `Agent` and `TimeSynchronizedAgent` as follows:

```
public interface Agent
extends Enactable, Messageable


public interface TimeSynchronizedAgent
extends TimeSynchronized, Agent
```

These two interfaces are accompanied by *abstract class*es called `AgentImpl` and `TimeSynchronizedAgentImpl` that provide skeleton implementations. Thus any class that extends `AgentImpl` or `TimeSynchronizedAgentImpl` and implements the core `execute` method can register, be executed, be synchronized with others, and/or send/receive messages.

5. EXPERIMENTS AND EVALUATION

Wa have performed three types of experiments and evaluations using MACE3J. Each of these is a domain-level experimental framework that runs under MACE3J and has been tested in several hardware/software environments including both sequential uniprocessor, cluster, and scalable multiprocessor machines.

TaskModel: TaskModel is a general, parameterizable agent-based distributed workflow framework, designed for organization modeling in general, and more specifically for research into the distributed compositional dynamics of organization, following [10]. A TaskModel workflow comprises a set of atomic *tasks* arranged in a directed graph of arbitrary topology. Each task is a specification of an activity that consumes a specifiable, asynchronously-arriving collection of task-specific input resource types and transforms them in a task-specific way to produce a related collection of task-specific output resource types. The output resource types are conditionally directed to other tasks as their inputs (the task-interdependency network may have choice points). A TaskModel thus specifies conditional relationships among *types* of activities, input resources, and output resources.

Agents are the primary loci of action in TaskModel. Each task is mapped to some agent using a *task-to-agent-mapping-process*; at any given time, an agent may handle zero or more tasks, and task allocation is potentially flexible and dynamic (cf. [10]. In execution, *problem instances* (PIs) are introduced into the TaskModel network, possibly in an asynchronous way, as input resources. Interpreting messages and processing task instances consumes resources; each agent's activities are constrained by the relationship between its *resource utilization function* (how much of what resource is required by each action type) and its *available resource base*. Agents receive and process messages containing input resources, executing the task instance corresponding to a specific PI when a) all of its PI-relevant inputs, and b) adequate resources, are available (a resource-bounded dataflow model). In this way, components of individual PIs 'travel' through the task and agent network. Multiple PIs can easily traverse the network in asynchronous and overlapping ways.

Agents exchange input and output resources (products) by sending messages. To support this messaging under dynamic task allocation, each agent must contain a database of knowledge about the task-agent mapping of tasks that are interdependent with its own, to compute where to send outputs and to validate inputs. In the spirit of [10] we call this information *organizational knowledge*. Viewed globally, the aggregated organizational knowledge captures the entire organizational topology in a distributed, overlapping, localized way.

Thus a 'minimal agent' in the TaskModel system has a) message receiving and sending facilities, b) one or more input message types, c) one or more output message types, d) a local task database containing the set of task (type) specifications currently allocated to the agent, e) a process and database for tracking asynchronous problem instance components and matching them to task types, f) a dynamic resource allocation system, g) a general task scheduling and execution procedure, and h) an organizational knowledge base.

Beyond this structure, the computational granularity of TaskModel tasks is arbitrary; more complex tasks and denser task-agent mapping make for larger-grain agents. Varying these two parameters provides a simple way of exploring MACE3J performance over a wide variety of agent granularity. Similarly, the number, type, frequency, etc. of tasks, agents, resources, and PIs can also be varied to experiment with changing performance demands, organizational reconfiguration, load response, etc. Finally, the TaskModel framework can be viewed as a multilevel organization model, with the task-interdependency network comprising one organization level, and the task-to-agent mapping and inter-agent message flows comprising a second, higher level. In this view, individual tasks may be declared to be *ActiveObject*s having them implement members of the MACE3J interface family discussed above, and agents may then become composites of lower-level active tasks. This illustrates the potential for MACE3J to flexibly support multi-level compositional frameworks (also a goal of [2]) using a range of implementations.

As a rough illustration of performance, we have run the TaskModel environment under MACE3J in experiments with 10,000 tasks mapped to 5,000 agents, and 10 million total messages exchanged, in under 4 minutes on a 900MhZ/500MB Athlon uniprocessor, Windows 2000, Sun JVM configuration. We've also run TaskModel/MACE3J in multiprocessing mode on the NCSA SGI Origin, but have not yet completed scaling or performance tests in this environment (future work).

Dynamic Information Quality Simulation System (DIQSS) The MACE3J/DIQSS experiment was designed to explore the ability of MACE3J to support a legacy agent application, and to exercise MACE3J's MVC and user-interface agent facilities. "Information quality dynamics" (IQD) theories attempt to explain how the overall quality profile of a large information base (such as a search engine's web index or a document repository) evolves as a result of accesses and interventions by a collection of agents over time [8]. Our original DIQSS was written in in Visual J++ in 1999 as a custom stand-alone simulation, with a hard-coded interface. It included a parameterized information-base model, several accessor and intervention agents, and several graphical displays of evolving IQ parameters. Not having worked with the DIQSS code in over two years, we ported it to MACE3J in under 3 hours with identical external functionality, including recreating the DIQSS user interfaces as asynchronously-running MACE3J agents.

Large-Grain Concurrency Model (LGCM) LGCM is a very simple multi-agent model designed solely to experiment with and validate the multiprocessor scaling behavior of MACE3J. LGCM creates a parameterizable number of large-grain agents with parameterizable complexity (>3 seconds cpu time per agent execution). We have run LGCM on the 1000-processor NCSA SGI Origin multicomputer in configurations of from one to 48 processors with one large-grain agent per process, and have demonstrated a) complete processor utilization and b) almost linear speedup for relatively independent (few messages) agents. We've duplicated this result in a 4-processor Sun cluster and on a dual-processor linux configuration at the GSLIS ISRL.

Overall, MACE3J has proven to be an effective and scalable testbed for a variety of implemented multi-agent systems. It addresses a number of important tradeoffs, and usefully extends the family of community-sharable tools for scientific research in MAS.

6. ACKNOWLEDGMENTS

This work was partially supported under NSF grant 99-81797, by a grant of time on the SGI Origin multiprocessor from the National Center for Supercomputer Applications (NCSA) at the University of Illinois, and by the Information Systems Research Lab (ISRL) of the Graduate School of Library and Information Science. We also thank Larry Jackson and Brynnen Owen for help with the Sun and Linux cluster experiments, Jeremy Nelson and Eric Rankin for discussions, help researching related systems, and early specification of some MACE3J interface components, Walt Scacchi for helpful discussions, and the referees for their helpful comments.

7. REFERENCES

[1] John Batali. Computational Simulations of the Emergence of Grammar. Chapter 24 in James R. Hurford, Michael Studdert- Kennedy, and Chris Knight, *Approaches to the Evolution of Language: Social and Cognitive Bases*, Cambridge University Press, 1998.

[2] Nick Collier. RePast: the REcursive Porous Agent Simulation Toolkit. http://repast.sourceforge.net (6/2001)

[3] James H. Cowie. Scalable Simulation Framework API Reference Manual, Version 1.0 Technical Report, Dartmouth SSF project and SSFNet, March, 1999.

[4] Keith Decker "Task Environment Centered Simulation" in [17].

[5] Thomas Eiter and Viviana Mascardi. Comparing Environments for Developing Software Agents. Technical Report INFSYS RR-1843-01-02 Knowledge Based Systems Group - E184/3, Institute of Information Systems Computer Science Department, Vienna University of Technology, Favoritenstrasse 11, A-1040 Vienna, Austria/Europe, March 2001.

[6] Les Gasser, Carl Braganza, and Nava Herman, "MACE: A Flexible Testbed for Distributed AI Research," in Michael N. Huhns, ed., Distributed Artificial Intelligence, Pitman Publishers, 1987, 119-152.

[7] Les Gasser and Jean-Pierre Briot. Object-Based Concurrent Computation and DAI. in N.M. Avouris and L. Gasser, eds., *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer Academic Publishers, 1992.

[8] Les Gasser and Besiki Stvilia "A New Theory of Information Quality", GSLIS Technical Report ISRN UIUCLIS–2001/1+AMAS, 2001.

[9] Les Gasser. MAS Infrastructure Definitions, Needs, Prospects. in T. Wagner and O Rana, (eds.), /em Infrastructure for Scalable Multi-agent Systems, Lecture Notes in Computer Science, Springer, 2001.

[10] Les Gasser and Toru Ishida. A Dynamic Organizational Architecture for Adaptive Problem Solving. *Proceedings of the 1991 National Conference on Artificial Intelligence*, 1991

[11] O. Gutknecht and J. Ferber. "MadKit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems" Research Report LIRMM 97188, Laboratoire d'informatique, de Robotique, et de Microelectronique de Montpellier, December 1997.

[12] Steven Hanks, Martha Pollack, and Paul R. Cohen. Benchmarks, Testbeds, Controlled Experimentation, and the Design of Agent Architectures. *AI Magazine*, 14(4) pp 17-42, Winter, 1993.

[13] T. Kielmann, P. Hatcher, L. Bouge, H.E. Bal. "Enabling Java for High-Performance Computing: Exploiting Distributed Shared Memory and Remote Method Invocation." CACM, 44(10) October, 2001.

[14] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture.* Prentice Hall, 1999.

[15] Brian Logan and Georgos Theodoropoulos. The Distributed Simulation of Multi-Agent Systems. *Proceedings of the IEEE*, Vol 89, No 2, February 2001, pp 174–185.

[16] Divine Ndumu, Hyacinth Nwana, Lyndon Lee and Haydn Haynes. Visualisation of Distributed Multi-Agent Systems in *Applied Artifical Intelligence*, Vol 13 (1), 1999, p187-208.

[17] M. Prietula, K. Carley, and L. Gasser. *Simulating Organizations: Computational Models of Institutions and Groups.* AAAI Press/MIT Press, 1998.

[18] Ramzi Suleiman, Nigel Gilbert, and Klaus G. Troitzsch. *Tools and Techniques for Social Science Simulation. Physica 2000*, Berlin, ISBN 3-7908-1265-X, 2000.

[19] Song Choi and Walt Scacchi. Modeling and Simulating Software Acquisition Process Architectures *Journal of Systems and Software*, 59(3) 343-354, 2001.

[20] Milind Tambe. Personal communication concerning implementation experiences using HLA/.RTI, 2001.

[21] P.E. Trapa and M.A. Nowak. Nash equilibria for an evolutionary language game. Journal of Mathematical Biology, 41(2):172–188, 2000.

[22] Regis Vincent, Bryan Horling, and Victor Lesser. An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator. in T. Wagner and O Rana, (eds.), /em Infrastructure for Scalable Multi-agent Systems, Lecture Notes in Computer Science, Springer, 2001.

APPENDIX

A. MACE3J SYSTEM REQUIREMENTS

Unix or Windows machine with internet connection (for downloads), Java 1.3 (Sun JDK preferred), a fast processor (>600Mhz preferred) and significant memory (256MB or more). Information on MACE3J availability can be found at `http://www.isrl.uiuc.edu/amag/` (2002).