LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-209617

Scalable Entity-Based Modeling of Population-Based Systems, Final LDRD Report

Andrew J. Cleary, Steven G. Smith, Tanya K. Vassilevska, David R. Jefferson

February 11, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

A Final Report for an LDRD Project for
Exploratory Research in the Disciplines

Tracking Code: 03-ERD-030

SCALABLE ENTITY-BASED MODELING OF POPULATION-BASED SYSTEMS

By Andrew J. Cleary, PI
Steven G. Smith
Tanya Kostova Vassilevska
David R. Jefferson

Center for Applied Scientific Computing, Computation Directorate

Abstract

The goal of this project has been to develop tools, capabilities and expertise in the modeling of complex population-based systems via scalable entity-based modeling (EBM). Our initial focal application domain has been the dynamics of large populations exposed to disease-causing agents, a topic of interest to the Department of Homeland Security in the context of bioterrorism. In the academic community, discrete simulation technology based on individual entities *has shown initial success, but the technology has not been scaled to the problem sizes or computational resources of LLNL. Our developmental emphasis has been on the extension of this technology to parallel computers and maturation of the technology from an academic to a lab setting.*

BACKGROUND AND MOTIVATION

One of the most important methodological advances within the DOE labs over the last decade has been the widespread embrace of large-scale computational modeling and simulation for the advancement of science and technology. The mission of the Center for Applied Scientific Computing (CASC) at LLNL includes research, development, and promulgation of such computational tools into the lab's programs. This project has been focused on developing scalable tools and expertise in a new discrete simulation and modeling paradigm that is applicable to the lab's rapidly growing programmatic interest in highly complex biological and population-based phenomena. The lab's interest in these systems has risen dramatically with the establishment of the Department of Homeland Security (DHS) and its nascent arm at LLNL, the Homeland Security Organization (HSO). With an appropriate set of simulation tools, large-scale computational modeling could be a powerful tool for analyzing these phenomena, and we anticipate similar needs arising throughout the lab in the future.

In the past, LLNL has focused on continuum-based simulation tools for physics applications. Implicit in continuum formulations are various types of averaging, which in some cases may fail to capture the macroscopic behavior of the system accurately. As a

critical example, it has been well documented in AIDS research [HyLiSt03] and now suspected in the recent SARS developments that disease spread may be largely driven by "super spreaders": patients that have infected 100% of their contacts, directly or indirectly, while many others have been far less infectious [AlMc03]. That is, a handful of statistical *outliers* are responsible for the majority of the spread of the pathogen. Simulation technologies based on averaging do not properly capture the effect of such outliers.

In the wider community, a class of discrete simulation technologies with roots in *cellular automata* [Wolfram02] and *general systems theory* have been used to model populations with highly variable individuals. The class is alternately called *agent-based modeling* or *individual-based modeling,* which we refer to collectively as *entity-based modeling (EBM)*. The core concept is to explicitly track *individuals* in a simulation and thus avoid the averaging reductionism that is appropriate in less complex systems. Public domain toolkits enabling simple simulations using this paradigm are experiencing rapidly growing user bases, but the technology is preliminary and needs to be extended in several ways before it can be used for the lab's problems.

TECHNICAL ACCOMPLISHMENTS AND MILESTONES

We have successfully run a basic epidemiological model on 1024 processors and have completed scaling studies for simulations with different communication-to-computation ratios that indicate preliminary success while identifying bottlenecks to further scalablity. We have developed several innovative architectural concepts that form a powerful scalable architecture for EBM, where that scalability is both in terms of parallel execution as well as application scalability. In addition, with the expansion of our project to the infectious disease domain, we completed development of reusable disease and immune system components.

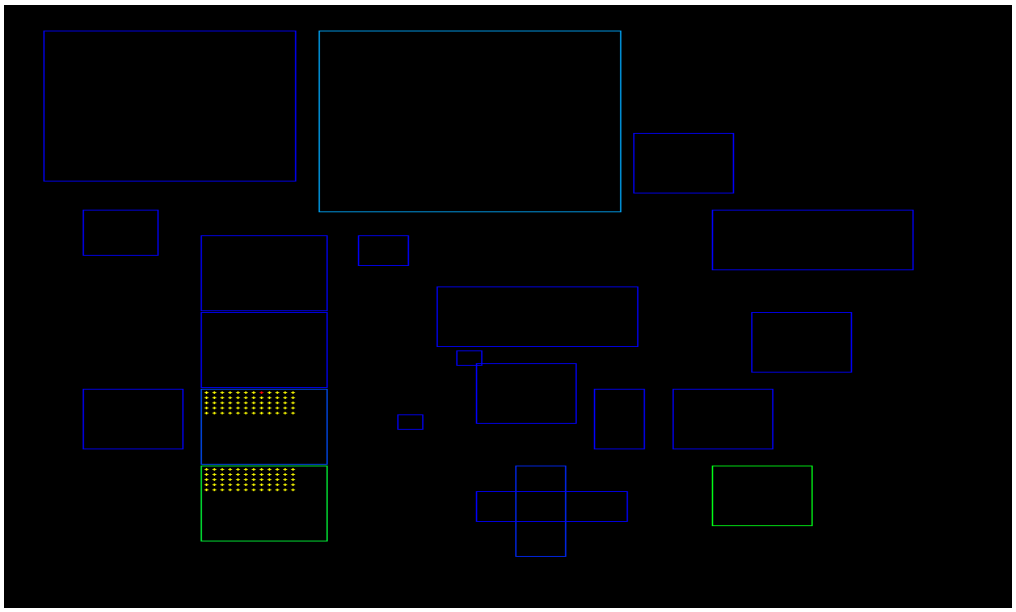


Figure 1: Screenshot from our model of Lackland AFB. Rectangles are locations on the base, color-coded with level of viral contamination. Dots are recruits, color-coded for current viral load. In this shot, recruits are in two classrooms, and there is contamination in one of those classrooms plus the medical facility.

Initial modeling and parallelization successes. Figure 1 is a screenshot from the model we produced for our first customer, the SPIDER project: a simulation of Air Force training recruits at Lackland Airforce Base, which has had endemic and costly problems with adenoviral infection. SPIDER used our entity-based model to fit the parameters of the disease transmission model for Lackland in the hopes of identifying causal mechanisms and prophylactic opportunities. Figure 2 shows the viral load of all of the recruits as a function of time. Using our models and simulations, SPIDER demonstrated the ability to differentiate between several scenarios, including a contaminated room as in Figure 2 versus recruits that enter training infected and thus infect everyone else.

We tackled several important issues in the development of our initial parallel infectious disease capability. The lab's parallel computing systems do not support parallel Java programs, requiring us to fashion our own support, though we were aided by the wide availability of public domain Java packages. To support a parallel scheduling capability, we developed an extended parallel version of Repast's Schedule class. As we detailed in our original proposal, this initial implementation parallelizes over the spatial domain of the simulation, thus requiring object migration when entities move from one processor's domain to another. To support this, we have implemented a remote migration component

and integrated it into our modeling package.

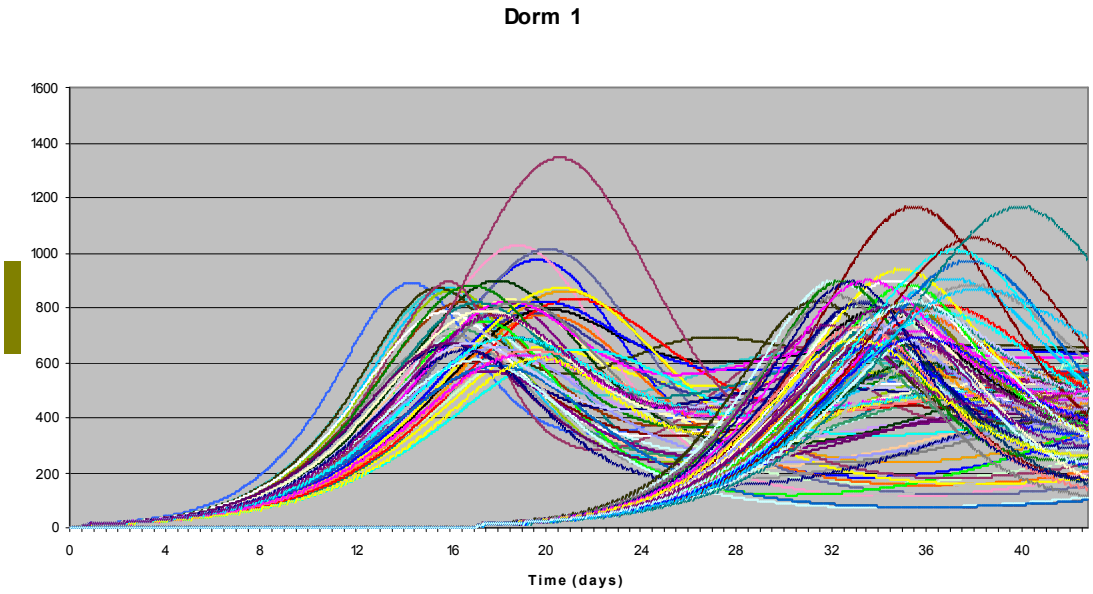


Figure 2: Graph of viral load for each recruit as a function of time. The graph structure serves as a signature for indicating different causal scenarios.

THE DEMONSTRATION APPLICATION AND SCALING EXPERIMENTS

We now present a scaling study of our migration package in the context of our epidemiological simulations.

A set of scaling experiments was conducted with a demonstration application in order to understand the scaling properties of the framework being developed. The demonstration application had the basic characteristics of the targeted smallpox vaccination application. The major difference was the use of a simplified movement model to enable control over the key scaling parameters (such as communication/computation ratio). The disease model was the same in both applications. The model is composed of people entities and location entities. Each entity in the model is represented as an object with state variables that control the disease model. The people also contain state to model their behavior. In the simplified case, the only behavior being simulated is movement between locations by a fixed schedule. The model is implemented in Java using the Repast infrastructure with additions to support parallelism consisting of a parallel nested event queue and an inter-processor communication framework to handle moving people between remote locations.

Virus Model

The virus model is defined by a set of equations governing the growth and decay of the virus for both people and locations. When a person is present in a location the person sheds virus into the location (if infected) and picks up virus from the location. The immune response model is of the form proposed by Anderson and May (2000).

Figure 3

Virus model for people:

If ($V_{person} < T_3$)

$d V_{person}/dt = V_{in} - \sigma * E * V_{person}$

$dE/dt = \Lambda - \mu * E + \varepsilon * E * V_{person}$

else

$d V_{person}/dt = V_{in} + r * V_{person} - \sigma * E * V_{person}$

$dE/dt = \Lambda - \mu * E + \varepsilon * E * V_{person}$

If ($V_{person} > T_2$)

$V_{out} = \alpha * V_{person}$

else

$V_{out} = 0.0$

Virus model for locations:

$dV_{location}/dt = \Sigma V_{out} - \gamma * V_{location}$

The equations controlling an individual's virus load (V_{person}) are given in figure 1. An individual is infectious (gives off virus in proportional to V_{person}) if the virus load is greater then the T_2 threshold. The T_3 threshold controls when the virus begins to replicate internally. The growth and death are controlled by 5 parameters: virus reproduction rate r, effector cells basic production rate Λ, effector cells reproduction induced by virus ε, effector cells mortality rate μ, effector cells induced virus elimination rate σ. The effector cells (E) are representative of a person's immune system. V_{in} and V_{out} represent the virus being picked up from the location and being shed. The amount of virus shed into a location is controlled by parameter α. The virus load for a location ($V_{location}$) is computed using the sum of the virus outputs for all people in the location and a decay rate γ. For the studies done in this paper, the virus model was updated at 5 minute intervals.

The Movement Model

For the demonstration application a fixed schedule movement model was used. Each person repeatedly visits a set of ten locations chosen at random at the start of the simulation and these locations are repeatedly visited until the simulation completes. The

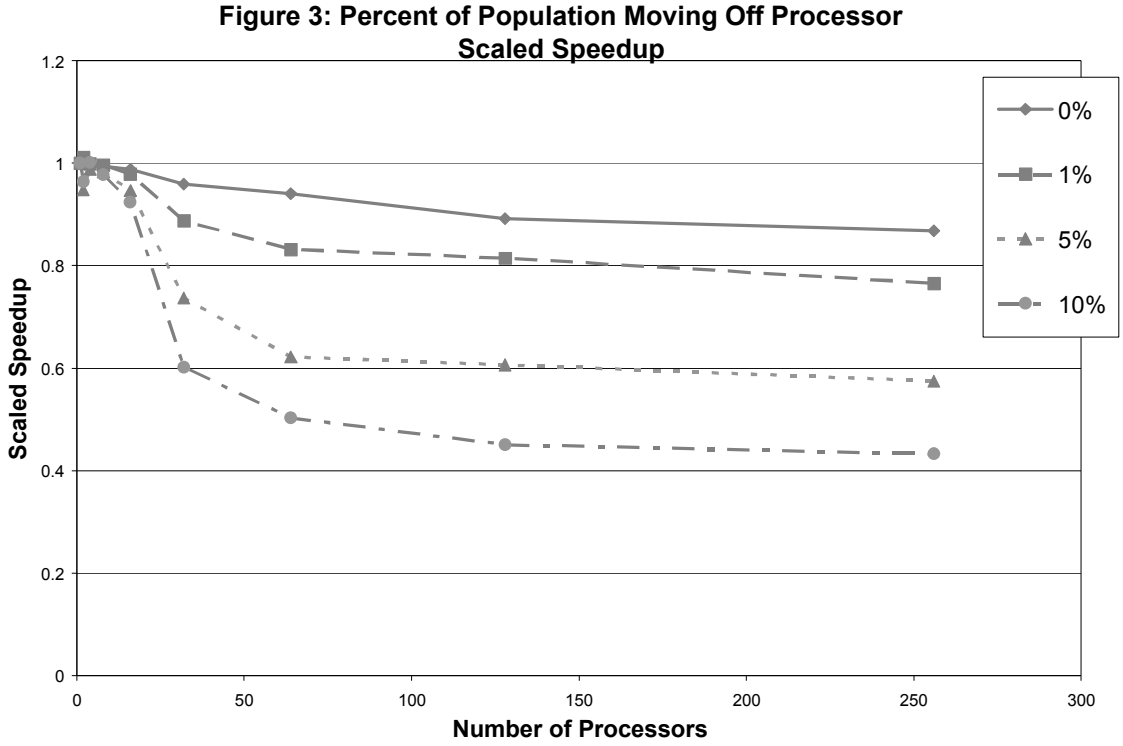
movement was selected to occur at 30 minute intervals. In order to understand the impact of changing the amount of communication occurring, the moves were selected to be either a location on the same processor or a location on a remote processor.

For the targeted smallpox application a more complicated behavior model was developed that incorporated medical treatment and vaccination facilities in order to study the impact of different vaccination strategies. When a person became infected they could alter their movement to stay at home or seek medical treatment. When an epidemic was indicated (e.g. first person reporting to a medical facility for treatment) an immunization strategy could go into effect and people would alter their movements to visit the immunization facilities. Effects on scalability of the vaccination behavior model were not studied.

Scaling Studies

Two performance studies were conducted. In the first experiment, a scaling study was performed using a higher frequency of movement than was planned for the smallpox vaccination study. The number of off-node moves was set to 10%. The results of the scaled speedup are shown in figure 4. The problem contains 50000 people and 10000 locations per processor. Due to moves being selected at random, at any given point in time the exact number of people on a processor will vary during a run. The largest simulation conducted represented 25.6 million people and 5.12 million locations.

A second experiment varied the percent of off-node moves to examine the impact of the amount of communication going on at each movement (figure 5). Since the schedule was selected at random there is very little locality preserved and, as a result, the communication is all-to-all. As expected, as the number of off-processor moves is decreased the scaling improves. For the 0% off-node move case, the scaling should be near ideal. The poor scaling in this case is being investigated; the initial inspection of synchronization points (e.g. maintaining a synchronized global simulation time) and other likely sources (e.g. I/O) did not reveal the source for the inefficiency.



Despite the anomalies and moderate scaling, it is believed the smallpox model would have scaled reasonably due to fewer moves and better locality in the schedules intended for the vaccination studies. The basic framework approach developed would need additional optimization work. For problems with greater communication requirements, different approaches would need to be investigated.

DESIGN OF REUSABLE SCALABLE SIMULATION INFRASTRUCTURE

One of the key technical achievements of this project has been the development of designs for two crucial layers of software that would serve as the foundation for a truly massively scalable framework for ABM. Because the project was cancelled prematurely, these designs were completed to varying level of detail, and only a few of the software components were implemented. Nonetheless, in the 1.5 years since the canceling of the project and the writing of this report, it is still the case that no tools for scalable ABM have been developed and published, and thus this design maintains relevance and promise as a path forward to massive ABM capability.

First we discuss out design of a scalable, hierarchical layer for DES, and then we follow with our design for ABM that will use the DES layer in its implementation.

DESIGN OF A NEW PARALLEL DES LAYER

We have developed a proposal for a hierarchical, object-oriented discrete event simulation capability as a result of design needs dictated by our development of a new framework for ABM. Each change we made from the more traditional Parallel DES (PADS) was driven either by changes in software technology that have occurred since the core of PADS development, by the needs of ABM, or by the needs of parallel scalability. The resulting DES capability can be used stand-alone as an alternative to the standard PADS model, though its intended use is in the implementation of our ABM layer. The main deliverable from this paper is a set of interfaces and their description, though we also present important use cases, most deriving from the needs of ABM, proposed classes that implement these interfaces, and notes on implementation of those classes. We have implemented some but not all of the interfaces and classes described below.

A review of the basic PADS conceptual model

The fundamental programming concepts in PADS systems such as TimeWarp {Jeff89] are the *event, schedule,* and the *logical process (LP).* LPs in TimeWarp are an abstraction for threads of execution, thus separating the logical notion of a thread from the physical notion of which processor each LP runs on. Each LP has *state,* and all simulation state must belong to some LP. The Schedule is a priority queue of Events, ordered by the "simulation time" associated with that event. Events in TimeWarp consist of a simulation time, a LP id, and parameters for a procedural function call to be made on the internal state associated with that LP. Simulation state can only change during the processing of an Event. Each Event may affect state only on its specified LP. New events may be generated during the processing of an event.

Programmers using TimeWarp to build a simulation were asked to decompose their problem into independent **processes**, which did not require them to explicitly consider parallelism, but implicitly exposed potential parallelism; it was up to the configuration stage to determine the actual parallel mapping. Events scheduled internal to a process were necessarily **local**; but events called on other processes might be remote, though if more than one LP ended up running on the same processor, an execution savings resulted (the "no one minds an unexpected improvement" rule). It is also worth noting a restriction of the TimeWarp system: the **only** method of communication between processes is events.

Overall, the PADS conceptual model provides a simple and low-level programming layer consisting of a small number of general and expressive concepts that apply to many situations, and yet that are constrained in their relationships so that reusable and efficient implementations can be provided. Similarly, we see our new DES layer as more of a programming layer than a simulation layer. The two most popular frameworks for ABM, Swarm and Repast, are both built on top of a DES model, and we follow this two-layered architecture. Like any programming layer, we describe our DES layer and some use cases, but it is up to the programmer to be sure that they use the components correctly.

Modifying the PADS model for object orientation

Our model differs from the traditional PADS model in several dimensions. Here, we describe changes to the PADS model spurred by the adoption in full of object-oriented methodology and technology.

In object oriented software, the key programming concept is the object rather than the procedural function. We thus seek to redefine the basic PADS concepts in terms of objects rather than procedures. Where the basic components of the PADS model are events, schedules, and logical processes, the basic components of our model are events, schedules, and objects. In some senses, the LP of the PADS model foreshadowed objects: an LP has both state and events are function calls that can affect that state. The basic definition of an object in OO is that it is a self-contained collection of state and method calls that can affect that state. Thus, the move from LP to object is a somewhat obvious one.

However, the transition from LPs to Objects brings several subtle changes that appear minor at first but manifest larger in the final conceptual model. One big difference is that in the PADS model, each LP "object" is what OO would call the same "class" (or at least, the same interface), since it is assumed in PADS that each LP supports the same function calls. In OO, different objects in general support (often wildly) different method calls. Also note there is an implicit assumption of granularity in the PADS model: it is assumed that an LP is approximately the granularity of a separate execution thread. It is somewhat typical to map each LP to a single processor in PADS, or perhaps a small handful of LPs to each processor. In contrast, our model may support objects of many different granularities, and a processor might in general have a very large collection of objects. Tied in with this is *recursion:* in general, objects in OO may be composed of other objects. LPs in the PADS model could not be composed of other LPs, the notion just did not make sense.

We see a similar difference in our model of events: they are now method calls on an *object* rather than procedural calls on a process. Like PADS events, an event in our model has a time associated with it used to order its execution on the schedule, as well as parameters for a method call. Here, the recursive nature of OO comes back in: the parameters themselves may be objects. In contrast, in PADS an LP could not be a parameter. The PADS notion of an LP was a somewhat static one: LPs could be moved for load balancing purposes, but they weren't an object in the simulation state per se'. In our model, like LPs, all simulation state must be contained in some object, but objects can have arbitrary relationships with each other. Like the PADS model, an event in our system is required to happen on the object at the simulation time on the processor that the object is on; each object is assumed to exist on a single processor (we defer until later a design for "parallel objects"), so that execution is implicitly assumed to happen on the processor on which the object resides, just as the PADS model insists that execution of an event occurs on the processor that holds the event's LP.

The smaller granularity of Objects vs LPs presents an implementational problem surrounding the notion of addressing. In the PADS model, an LP can schedule an event targeting any other LP, from which it follows that each LP must know the processor location of all targetable LPs. If we were to extend this in a straightforward way, it would imply that each object must know the processor location of all objects in the simulation, an unrealistic burden even in a setting of only a few processors and an absolute block to scalability of any significant sort (in fact, experience in many domains has shown that parallel systems that require each processor to maintain O(P) information are inherently non-scalable), and thus some sort of limited and thus manageable object addressing scheme is needed.

To manage the number of objects that can be addressed in parallel events, we introduce the notion of a *namespace*. Note first that we assume the existence of some form of remote addressing protocol without needing to be specific about its implementation, e.g., in the PADS model, some mechanism for referencing remote LPs must exist and we need a similar mechanism here (a simple mechanism is to use strings). A *namespace* is essentially a registry of objects that may be addressed. An object that is remotely addressable by events must be registered with the namespace using its remote address. In this way, the simulation writer can limit the number of remotely addressable objects from a given processor to a manageable number by utilizing a namespace that spans only a small fraction of all of the processors, thus achieving the locality needed for scalability.

Up to this point, we have not discussed our notion of the Schedule explicitly. Until we discuss hierarchy in the sequel, our concept of the Schedule is fairly consistent with the PADS concept. The execution of a DES is basically a continuous loop in which the next event(s) on the Schedule are executed, potentially resulting in the addition of more events, until the simulation has ended. In PADS, there is an explicit global schedule, and then an implicit local schedule for each LP. We make this notion explicit: we have a Schedule interface, with both sequential and parallel implementations. The first parallel implementation we have uses the sequential implementation as part of its implementation. This simple parallel schedule has a synchronized execution: the processors collectively find the next event time across all processors' local schedules, and then events at that time are executed. Obviously, the only source of parallelism in this implementation is events that are scheduled for execution at the same time, and while we have plans to implement a more asynchronous version later, when combined with some of our other concepts this synchronized schedule is still quite useful.

The main semantic difference (until we introduce hierarchy) between our Schedules and a PADS schedule is that ours can have one or more namespaces associated with them, and events on Schedules can either use normal Java references to address targets that are necessarily on the same processor, or they can use remote addresses registered with a namespace to address potentially remote objects.

There is a semantic and implementational complication that arises from the possibility that an object may be simultaneously a "target" of a remote event and a parameter in another remote event: the event that targets that object is supposed to be executed on the

processor on which that object resides, but since the object is a parameter in another event, the object does not really reside on any particular processor until that other event is processed. Put another way, the namespace in which that object is registered is required to manage the association of that object and its processor, but it cannot do so while the object is "in transition" so to speak. There are a variety of increasingly complex ways to define the semantics of this sort of situation and to thus implement it in code, but we take a cue from the PADS model to present the simplest and least general of these on the assumption that many useful situations can be handled, and those that cannot will have to be deferred to future iterations of more complex and difficult support code. In the PADS model, LPs are the *remotely addressable objects* (RAOs), and they cannot move. For our first iteration, we make a comparable requirement: objects registered with a namespace as RAOs cannot be used as parameters in remote events, meaning that they cannot move off-processor. In fact, while at this point we have not required this, there is potential merit to an even further restriction: the only possible remote events are "move" events in which one object is moved from one containing object to another. We are experimenting with this restriction to ascertain its usefulness and it is a particularly good match to our ABM design where containment features prominently, and it makes implementation of the remote event mechanism particularly easy since no support is required for arbitrary method calls. However, requiring it in our DES model would require the introduction of the semantic concept of containment and for the time being we believe that that is too high-level a concept to introduce to the relatively low-level abstractions in DES, which we see more as a programming layer than a simulation layer.

The introduction of hierarchy

We introduce explicit and implicit notions of hierarchy into our DES model for several complementary reasons: parallel scalability, software extensibility and flexibility, and multiscale modeling.

Hierarchy appears in at least three places in our new model: object containment, nested schedules, and implicitly through nested schedules and nested namespaces. We have already introduced the notion of objects being contained in other objects. As we stated before, we save a more complete discussion of object containment for our ABM framework design because we see it more as a simulation concept, so we will not discuss it in further detail here.

Here we discuss the concept of nested namespaces. As we mentioned before, in the PADS model, each LP must know which processor every other LP is associated with. On the reasonable assumption that the number of processors and LPs scale together, this implies an overall memory requirement of O(P^2), and we submit without proof that any scheme that requires O(P^2) storage is not scalable. From this, it follows that in a scalable simulation, knowledge of each RAO may only be stored on some subset of processors. In our DES framework, this is accomplished by allowing the simulation writer to associate each namespace with a subset of processors only.

Since a namespace in turn is associated with a Schedule, this implies that schedules in our scheme may be associated with some subset of processors, and that by extension, there may in general be many more than just a single schedule in a simulation. It is through this mechanism that the concept of hierarchy really distinguishes our model from the PADS model. In the PADS model, there is one global schedule and an implicit local schedule on each processor used in the implementation of the global schedule, but it is a static and inflexible relationship between the global and local schedules. Rather than take a "global" approach to the issue of scheduling, we have remained consistent to our architectural notion that all software components are peers and that no single component is assumed to be global, and we thus envision many separate schedules throughout a complex simulation.

The key concept for coordinating different schedules in a single simulation is *nesting:* schedules can be linked into a parent/child relationship through a small number of simple additions to the Schedule interface hierarchy along with an intuitive semantic definition of nesting. An *EmbeddableSchedule* is a schedule that can be configured as a child to a parent schedule that implements the *NestedSchedule* interface (and a single schedule class can implement both interfaces if so desired). Each interface exposes Get/Set functions to establish the parent/child relationship. In addition, the EmbeddableSchedule interface defines method(s) that allow the parent schedule to control the execution of its children, i.e. ExecuteToTime(double time). This is a slight departure from the typical PADS definition, in which a Schedule is generally assumed to execute a continuous loop until the simulation is over. This functionality is used by the parent schedule to provide a standard (set of) semantic relationship(s) between a parent schedule and a child schedule, and we capture this by providing a basic implementation of a NestedSchedule. The semantics are captured in this psuedo-code fragment from our BasicSchedule implementation:

```
Class BasicSchedule implements EmbeddableSchedule implements NestedSchedule {
ExecuteToTime( double time )
{
  Loop until NextEventTime() > time:
    Loop over I=children schedules:
      I.ExecuteToTime( NextEventTime() );
    End Loop
  End Loop
}
}
```

Here, the function NextEventTime() returns the execution time of the next event on the schedule. Like the standard schedule loop, a BasicSchedule repeatedly grabs the next event on its queue and processes it. However, either before or after doing so (we allow for either variation though we have illustrated only one of those variants), it instructs its children schedules to "catch up".

There are a number of variants to the exact semantics of the parent/child relationship that can be captured either as inherited classes or configuration options: how "ties" between the parent and child are handled; whether children are processed before or after the parent; whether children are told to catch up exactly to the parent's time or rather some increment before or after that; and whether children are told to catch up in terms of execution time or rather the scheduling of new events (e.g., a child can be told "catch up to time X" or "catch up until you are sure you will not schedule any events until after time X"). We consider these second-order details in our architecture and we will not focus on them in this document, though the tradeoffs are an area for future research.

We have used the terminology "parent" and "child" purposefully to evoke the concept of a **tree** of schedules, as that is exactly what we intend. Tree structured simulations appear occasionally in the literature [Logan and Theodoropoulos 2000]. Here, the authors advocate a dynamic tree structure adjusted to reflect "spheres of influence". Essentially, the concept is to identify dynamically parts of the statespace of the simulation that directly affect each other through events and collocate them in the same part of the tree. This kind of dynamic, automatic clustering is difficult to identify and is generally expensive both in runtime execution cost and development cost to support even when it does work. In contrast, we take a pragmatic, maximum "bang for buck" approach: we require that the tree be identified by the simulation writer in the definition of the simulation itself, and take advantage of the structure that they create. They express these relationships through the construction of the tree of nested schedules. Whether the tree is constructed automatically and dynamically or statically by the simulation writer, the tree expresses and defines a hierarchy of locality: the root of the tree represents events that must be coordinated across all processors, while lower levels of the tree are successively more local. Intuitively, this is what we want: we want most of the action in the simulation to occur locally, and only a small amount to occur globally because otherwise that global interaction becomes a scalability bottleneck. The tree is the methodology for expressing these cascading scales of locality.

This is a natural extension of the PADS model, where each LP has a local schedule in service to a global schedule and each LP is a "virtual thread of execution". We retain this notion that a Schedule essentially fills the role of a "virtual thread of execution", but we generalize the possible set of threads of execution. Our architecture enables the simulation writer to construct trees of schedules (and thus virtual threads of execution) of arbitrary complexity and shape through recursion: any Schedule that implements both the EmbeddableSchedule and NestedSchedule interfaces (as our BasicSchedule does) can serve as an interior node in a tree of schedules, as it can be both a child to another schedule and have children of its own. Our ABM paper provides extended examples of using this crucial feature of our architecture in the setting of a full simulation. Here, we discuss some aspects of implementation and performance.

Look-ahead

Fujimoto has summarized the performance requirement of conservative PADS schemes thusly: "To get parallel performance, look-ahead is necessary." Look-ahead is defined in terms of a local schedule: if the local schedule is at time T but can be sure that the only events that will be scheduled on it between T and some future time T' will be strictly as a result of the processing of events on that schedule, then that schedule is safe (that is, no rollbacks) to execute without communication with other schedules until time T'. Much of the literature of PADS is involved with the problem of extracting knowledge of look-ahead, as this provides the units of parallel execution: a schedule with a safe lookahead time can execute independently of other schedules for the duration of the look-ahead. There are some automatic schemes for dynamically determining look-ahead, e.g. the spheres of influence idea cited earlier, along with syntactic addition that the simulation writer can use to indicate to the simulation engine opportunities for lookahead, e.g., in some variants of PADS, the pattern of communication between LPs must be static and then various forms of signals can be used to indicate look-ahead. In our architecture, we introduce new notions of look-ahead expression. This begins with our extended definition of event times.

We make the following observation about event times: there exist several possible legitimate and largely separate uses for the time assigned to an event. One use of an event time is to *order* events, or in general, to properly respect causality. A second sense of an event's time is to properly account for *duration,* that is, the processing of an event may represent the end of some process, and properly processing the event may require knowing the duration of that process. The first is strictly used for ordering of execution, while the second is used for accuracy of calculations. A third notion of time that is not entirely independent of the other two is to know the time at which the event was scheduled: this time might have causality implications and it may also be related to the duration of the process that created the event.

The basic PADS model includes only a single time, appropriate for the modeling situation in which the same time serves all purposes. We prefer to retain more flexibility in the interpretation of various definitions of time, and thus we associate 3 separate times with each event: the time at which the event was scheduled, the scheduled time plus the duration, and a time used for ordering the execution of events. At this point, we are deliberately somewhat fuzzy about these definitions because it is possible for different simulation systems to interpret them differently. However, we have implemented a specific definition in conjunction with our ABM architecture, which we now discuss.

Lamport [Lamport 1978] formalized the discussion of causality in DES by borrowing the concept of "space-time" graphs from the physics of relativity to illustrate that in general, an absolute ordering of events is not necessary and even somewhat arbitrary; what is important is that the *partial ordering* of events that dictates causality be adhered to. The concept that is important for parallel processing is that if events A and B happen close enough in time and far enough in distance that the intersection of their respective direct causal influences is empty, then the order in which we process A and B is irrelevant, and thus we can say that "A and B can be processed in parallel". For this to happen, the simulation writer must express the independence of these events. One way to express

independence of events is to schedule them at the same execution time, while retaining a second and possibly third notion of "event time" that can be used for accurate calculations. It is worthwhile noting that this concept has been suggested in several contexts, including Repast: the design documentation for Repast [Collier, 2000] says "units of time are merely a way to order the execution of events relative to each other", though most of the examples we have seen of simulations built using Repast use the time for more than just ordering.

Using the extra times associated with our events provides one form of look-ahead. In our ABM architecture, we use one time as the "execution time", and a second time as the "exact execution time". Along with the third time, the time at which the event was scheduled, the second time gives the duration, but we prefer to list it in this form because it has the intuitive interpretation as an "exact time at which the event would be processed in an exact simulation" to go along with the an interpretation of the first time as "the inexact time at which the event was processed." These definitions appear overly awkward, but they are consistent with notions of accuracy that we use at length in our ABM architecture and so we introduce them here. Note that this gives a way for the simulation writer to express parallelism: by co-scheduling events to have the same execution time. In terms of look-ahead, the expression of look-ahead here is the difference between the exact time and the execution time: even though two events may have different exact times, if they are given the same execution time, it is an expression that there cannot be a causal link between one and the other in the time difference between their exact times.

Nested schedules provide an alternate though complementary method to express lookahead. The exact form depends on exactly how the nested schedules are configured, but the general flavor is the same: whenever a parent schedule tells a child schedule to execute, it is telling the child schedule that it has a safe window of look-ahead. It is important that the simulation writer understand the semantics here to avoid unintended errors. The canonical example is a parent parallel schedule that has sequential children schedules. Assume that the simulation writer schedules local events on a local schedule, and potentially parallel events on the parent parallel schedule. When the parallel schedule is at time T and tells the children schedules to catchup, the semantics of the parent schedule are still in force: when a schedule is at time T, no new events can be added **before** T. Because of this, the parent schedule knows that it is safe for the children to advance to T. The parent schedule is using its knowledge of its children to know that none of the children can send an event to each other until time T and thus instructing them that they have a safe look-ahead window.

Time Windowing

Here we discuss several variants of a new API concept that we call *time-windowing* that is intended to promote parallel performance. These concepts are particularly well matched to our ABM infrastructure development under the topic of "approximate simulation" but we discuss them here as stand-alone concepts. We have laid the groundwork for this in the inclusion of multiple times in our events. The basis of the

ideas comes from combining Lamport's observation about only needing to respect a partial ordering of events with the look-ahead and coscheduling concepts necessary for parallel performance. The crux of the concept is to treat the **execution time** of an event as something with some wiggle room in it while retaining the notion of the **exact time** of the event.

Assume for example a set of events E(i), I=1..n, all of whose exact times are within some interval delta but are far enough in distance that they cannot effect each other; in Lamport's terminology, the relative ordering of the execution of these events is irrelevant, because there is no space-time line that can connect the events. We call this a time-window in which the enclosed events do not have a causal relationship. These events can be processed in parallel. If the simulation writer has constructed the simulation in a way that he knows these events have no causal connection, he needs an API by which to tell the parallel Schedule this information.

One method for a simulation writer to take advantage of a time-window is to use the separate event times explicitly: schedule events that can be processed in parallel at the same execution time, but use the exact time in all calculations. This method is simple and intuitive, but it is requires some coordination and agreement amongst the components that schedule events and so is less extensible and amenable to separate development of simulation components.

A more extensible approach is to create new APIs for Schedules that allow a simulation writer to control the grouping of events dynamically and externally, but the method carries more risks as well and so must be used carefully. The simplest form of this API is to set a TimeWindowSize parameter in units of time from some external software component (e.g. a GUI or the core Simulation object) and to interpret the exact time of an event as its *intended time* and to then allow the Schedule to set the execution time as the *actual time* that the event ended up being processed. Clearly, the semantic danger here results from the fact that the Schedule ultimately makes the determination of the actual execution time rather than the simulation writer, and thus the simulation writer must understand the implications of this to avoid semantic errors.

There are two different semantic interpretations of this time interval which we support separately. Both change the fundamental loop of the Schedule class from "execute in any order all events scheduled at the time of the first event on the schedule" to "execute in any order all events scheduled during the next interval of the requested size"; the difference lies in differing semantic restrictions on the scheduling of new events during the processing of a time-window. In the ApproxSim variant, new events can be added to the schedule at any time after the beginning of the current time window, including times **before** the end of that time window. In the SteppedSim variant, new events may only be added **after** the end of the current time window. We now discuss some of the implications of these two variants.

The SteppedSim variant is inherently more dangerous than ApproxSim, but it only applies when a strict look-ahead of size the time of the TimeWindow (or larger) exists, that is, if the processing of an event cannot result in another event for some bounded time, that time serves as a safe TimeWindow. We call this the SteppedSim variant because it is inspired by the common practice in many ABM simulations of scheduling events at regular time-intervals as a simplified control mechanism and the observation that such events lend themselves to parallel execution. SteppedSim can fail if the simulation has some events with small look-ahead, because causal connections can be lost.

There is another subtle and important difference between our extended model of event execution and the standard PADS model: determinacy. The strictest PADS models assume that either events are never coscheduled and thus form a complete ordering, or that any tiebreaker scheme is deterministic. The issue becomes important in the case that two coscheduled events effect the same part of the state-space of the simulation in a non-commutative way, because the result of the execution of those events then becomes order-dependent. There are a lot of strong arguments for determinacy, particularly debugging but also scientific repeatability, but such determinacy severely hampers performance. As well, we extend the concept of Lamport's partial ordering in the context of the many sources of approximation and error that are extant in a simulation (e.g. initial conditions are not known to infinite precision, etc) to observe that often, the notion that two events that are very closely scheduled in time necessarily occur in that order is a red herring similar to carrying nonsignificant digits in a numerical calculation: it is an overly strong interpretation of the data. So, while strict determinacy is convenient, it is actually the opposite of reality, and thus retaining it is unnecessary. Our use of the term "ApproxSim" is intentional and comes from this line of reasoning. Note that we could probably add a deterministic mode of execution that would in general severely hamper performance, but we do not discuss this enhancement further in this paper. We also note that a large amount of indeterminacy can be eliminated by setting the time window to zero; this strictly orders all events that are not exactly coscheduled. The interpretation here is that the TimeWindowing concept is a form of approximation that can be dialed up or down according to need, at least within some limits.

The ApproxSim variant trades safety for performance when compared to SteppedSim, though it still must be used with caution. Events can schedule new events with arbitrarily small look-ahead relative to the TimeWindow, but causality relationships must still be carefully considered by the simulation writer. If events E1 and E2 with $E1 < E2$ are processed in the same window and E1 creates an event $E3 < E2$ that has a causal effect on E2, the simulation will in general incur an error. Note however that if E3 has a lookahead greater than the TimeWindow, then execution is still consistent. The ApproxSim variant can tolerate some events with small look-aheads which SteppedSim cannot, but it cannot tolerate a chain of them. The ApproxSim does not offer as much opportunity for performance, however, because the end of the next window cannot be known until the

current window is processed, whereas SteppedSim knows this ahead of time and implementations can take advantage of this in terms of synchronization and communication.

The major difference between the windowed schedule approach to time windowing versus the use of coscheduled events is that the determination of the time window is decoupled from the implementation of the event-scheduling components of the simulation. Indeed, since it is the schedule that sets the execution times of events, events can be scheduled using exact times and the determination of execution time can be deferred to runtime. There are a number of attractive uses of this functionality. One implication is that components written by different authors are easier to compose into the same simulation since no prior agreement on step sizes needs to be incorporated into the source for those components. In fact, a component can be written assuming that it will execute in exact time semantics but still be used in a time windowing context without even knowing about time windowing. Of course, the composer of a simulation must be careful when using such a component to be sure that it fits the requirements of look-ahead. This can be done in a variety of ways, e.g. documentation or design-by-contract macros, and we leave the exact mechanism as a future detail.

The combination of time-windowing and nested schedules creates extended capabilities for a simulation author that we have not yet fully catalogued. We present an illustrative but non-comprehensive example, and we hope that extended experience using these software components will allow us to more formally catalog them.

Consider a situation in which there are two distinct types of events Y1 and Y2, each with their own characteristic look-ahead times T1 > T2. A common interpretation of look-ahead is that an event of type Y1 is the culmination of some minimal length process; that minimal length T1 then defines a look-ahead for the creation of events of type Y1 by any other event. At first glance, this creates a problem on a time-windowed schedule: if T1 is used as the time-window, causality errors can occur with the Y2 events; but using the T2 as the time-window restricts available performance enhancement in the processing of the Y1 events.

We can address this problem by using more than one schedule, one for each type of event, with each having a time-window corresponding to their look-ahead time. We set the schedule corresponding to the longer time window as the parent of the schedule with the shorter time window (this is a general rule of thumb for nesting schedules that we have seen in several contexts, and our tendency now is to try this idea first). The general flavor of the execution loop will then be a doubly nested loop of length T1 on the outer loop and T2 on the inner loop: at each outer loop, the parent schedule executes all Y1 events in some interval of size T1, creating events of type Y1 in the next interval and events of type Y2 in the current or future intervals along the way. Then, the child schedule is advanced in increments of size T2 to catch up with the parent, processing

events of type Y2 added previously, added by the just-completed execution of T1 events, or added during the execution of the current iterations of the child schedule, and possibly adding events of type Y1 to the next interval. Intuitively, what we are taking advantage of here is the long process that creates Y1 events to "preprocess" the Y1s that we already know about and defer thinking about other Y1s until processes that have not started yet may create Y1 events. We cannot do this with the Y2 events as easily, because they are created by a short process, and so we must process them closer to strict sequentiality.

The resulting execution achieves the execution savings of the longer T1 events for those events while respecting causality restraints: any process started by the execution of a T2 event cannot possibly end (thus creating a T1 event) until the next, unprocessed, T1 window.

As we noted, this is only an illustrative example, and there are certainly caveats as well as possibly other ways to achieve similar effects. For instance, in this case, it is equivalent to make both schedules children of a third schedule with "no-op" events scheduled every T1 seconds, and it may even make more sense to make the schedules siblings rather than parent/children. It is our intention to catalog uses of our components rather than dictating any particular use of them.

Proposed APIs/Code

All set methods have corresponding get events that we do not list for brevity.

Basic Object-Oriented Model

```
public interface Schedule
{

   public void setNameSpace ( NameSpace nameSpace );

   public void addEvent ( Event event );

   public void execute ( );

   public double getLastEventTime ( );

   public double getNextEventTime ( );



}

public interface Event
{

   public void setExactTime ( double time );

   public void setExecutionTime ( double time );

   public void setStartTime ( double time );

   public void setGlobalTarget ( RemotelyAddressableObjectReference target );

   public void setLocalTarget ( Object target );
```

```
   public void execute( ); /* should only be called by the Schedule */

   /* Classes implementing Event will add constructors for adding type-specific
       parameters to be used in the body of "execute" to call a function on the
       target object. */
   /* A partially implemented abstract class can be implemented simply to handle
      the time variables and then extended by inheritance for specific types of events */

}

public interface RemotelyAddressableObjectReference
{

   /* Internal methods used by the namespace classes; other classes use this interface
      as a marker interface */
}


public interface NameSpace
{

   public void Register ( RemotelyAddressableObjectReference object );

   public Object IsLocal ( RemotelyAddressableObjectReference object );
   /* Returns null if object is not local, otherwise returns local address */

   public int IsRemote ( RemotelyAddressableObjectReference object );
   /* Returns processor number of processor of object if namespace is parallel;
      returns negative value if namespace is sequential. However, suggested usage
      is to query "IsLocal" first, and if that returns a non-null, IsRemote should
      not be called, so that negative return values should not be needed. */

}
```

Hierarchy

```
public interface EmbeddableSchedule extends Schedule
{

   public void executeToTime ( double stopTime );

   public void setParent ( NestedSchedule schedule );
}


public interface NestedSchedule extends Schedule
{

   public void addChildSchedule ( EmbeddableSchedule subSchedule );
}
```

Time Windowing

```
public interface SteppedSchedule extends Schedule
{
   public void setTimeWindowSize ( double time );
}

public class ApproxSchedule extends Schedule
{
   public void setTimeWindowSize ( double time );
}
```

AN OBJECT-ORIENTED CONCEPTUAL MODEL FOR AGENT-BASED MODELING OF COMPLEX SYSTEMS

Here we detail our development of a conceptual model for representing the elements and their relationships and interactions extant in an *Agent-Based Model (ABM)* of a complex system. We detail several tangible benefits of this design versus existing designs (as embodied mainly in software packages such as REPAST and SWARM): our new conceptual model is a closer match to reality, resulting in faster and less error-prone implementation of models; it promotes composability and thus software reuse; and it maps well to parallel execution, particularly on the scalable object-oriented Discrete Event Simulation (DES) layer that we was detailed in the previous section. The result of completion of these two layers of software will be a relatively simple to implement and use and yet scalable and composable framework for massively parallel Agent-Based Models that will enable their adoption in the analysis of complex systems in many disciplines, including economics, biology, and social science.

Motivation of our ABM model

The field of Agent-based modeling and simulation are quite different from that of other simulation technologies. Attempting to use existing packages with minor tweaks is really like shoving a round peg into a square hole, and yet this is the approach that the major existing ABM software frameworks take. Important concepts in an ABM model can be hard (or even impossible) to find in these frameworks, and the central concepts in these frameworks are often leftovers from previous technology. In addition, the existing packages do not lend themselves well to parallel execution, particularly in a massively parallel setting. Given modern scalable parallel computers, a better understanding of the ABM space, and the fast pace of changes in the software engineering world, there is a niche for a next-generation reusable ABM software infrastructure that this document attempts to fill.

ABM are used in the study of a different class of phenomenological system than more common simulation technologies such as DES and PDE-based simulation. ABM arose mostly in the context of the study of so-called *complex adaptive systems* (CAS). Three major differences we note here are that CAS are typically not amenable to intuition and/or the state-of-the-art of scientific understanding of these phenomena is so preliminary that often the desired result of a simulation study is merely an intuitive, qualitative understanding of the system; CAS typically feature prominently a population of discrete *decision making* entities which brings a need for new semantic concepts into simulations; and CAS are often very "messy" systems in the sense that stochasticity, nonlinearity, and uncertainty are ubiquitous throughout the system, requiring that the simulation technology incorporate these notions in a natural way.

Characterizing our target class of systems has a more pragmatic end than justifying new simulation technology: we take advantage of commonalities of these systems as the key to providing software that is more reusable, efficient, and easy to use than trying to

retrofit existing code bases to ABMs. Our three commonalities are a common set of parts that makes up these systems (a common object model in OO-speak), a common language for expressing them (a standard set of interfaces), and common locality properties (common runtime traits). The common language provides ease of expression and thus ease of use for a simulation author. The identification of common parts and their relationships provides a common structure that can be written once in a reusable software framework and then reused across many different simulations by disseminating the framework to interested users. The language and structure together provide necessary though not sufficient support for composability and interoperability, that is, the ability to reuse simulation parts like environments and agents (described later) in simulations other than the one for which they were originally intended with little or no modification. Finally, the identification of common runtime properties such as characterization of locality can be used to produce algorithms and codes optimized for those common properties, particularly in the realm of parallelization. We detail our attempts to take advantage of these commonalities of our target class of applications throughout the rest of this paper.

Agency, Games, and Complex Systems

Here we delve deeper into our characterization of our target applications, thus setting the table for subsequent development of the design of our infrastructure. It is beyond the scope of this paper to review CAS at any great length, but we need to list some representative examples to help illustrate our characterization. Loosely, the types of system and phenomena that either fall into the CAS class or have lent themselves to ABM fall into three somewhat overlapping categories: biological, including ecosystems and evolution; human-based or designed systems, including economics and sociology as well as human-engineered systems like mobile phone networks; and complex nonlinear physical systems such as fluids and self-assembling complex molecules.

Each of these systems contains populations of distinct entities, which is an uneasy match at best to continuous simulation technologies based on differential equations. Continuous formulations can be used to address these phenomena, but they must treat the populations as large aggregates, and if individual differences and behavior are important to the overall evolution of the system, the continuous formulations may struggle to represent them. On the other side of the simulation spectrum lies discrete simulation technologies such as DES, N-body simulations, and particle-based simulations. These types of simulation can address the issue of separate representation of each entity, but they implicitly assume a certain amount of simplicity about those entities. One way to describe this is that entities in these other technologies are **reactive:** they make no changes to their operating behavior until they are acted upon, or at the most, they undergo a regular and usually smooth change. Most electrical components in a circuit simulation will not change its state on its own, and rather must be acted upon by some outside force, whether it be a voltage change, a current change, a flip of a switch, etc. At the very least, those electrical components that do evolve over time do so in a smooth and simply predictable way, e.g. a discharging capacitor follows a well-known formula for doing so. Looking at our list of example systems, we can see that reactivy is not a good match: humans **make decisions,**

animals in an ecosystem **respond to internally evolving needs,** atoms in a folding protein **search** for a lowest energy state, etc. We capture this more proactive nature of entities as a concept we call *agency.*

In his book *Investigations,* Stuart Kauffman [Kau01] presents a thought-provoking body of work on the concept of agency and agents, and defines an agent roughly as "an entity that manipulates its relationship to its environment to produce useful work-cycles." Our definition of agency has a similar feel, but focuses on the term "useful" to give a definition in terms of mathematical algorithms: "an agent is an entity that manipulates its relationship to its environment to optimize some internal objective function." Here, a "useful" work-cycle is seen to be one that advances the optimization of an internal objective function. In an economic system for example, the objective functions are typically concepts such as "maximize my total worth" or "minimize my risk"; in an evolutionary context, the objective is "maximize representation of my genes in future generations" [Dawkins77].

There is another way to interpret these systems that is inspired by game theory that is useful for elucidation. Because our agents have agency and internal objective functions that they are trying to optimize, these systems can be viewed as *multi-player games* in the game-theoretic sense, with each agent playing the role of an autonomous (at runtime at least) player. Indeed, one of the canonical examples of systems analyzed through existing ABM tools is the *Prisoner's Dilemma* (PD) in various forms, including multiplayer. The PD problem is a classic game-theoretic problem, and while there are several formulations, the basic concept is that two prisoners are captured and held separately, and each is offered a deal by their captors so that their choices are to *defect,* that is, to turn against their fellow prisoner, or to *cooperate*, to stay loyal to their fellow prisoner. The two-player form of the PD is a classic problem for analytical game theory: depending on the rewards and punishments associated with the various combinations of defecting and cooperating, different optimal strategies can be calculated for the prisoners. The problem quickly becomes intractable analytically, however, if complicating factors are included: uncertainty, repetition, multiple players, memory, etc. For this reason, the more sophisticated versions of the PD were one of the driving applications behind several early ABM simulation framework efforts. Note that another term for an action in game setting is a *move.* We include this concept in our conceptual model as the set of actions that an entity with agency can undertake. In the PD for example, the allowable "moves" are "cooperate" or "defect", but the allowable moves in an ABM will in general be a function of the system being modeled.

The many variants of PD illustrate an important difference between the CAS class of applications and those that lend themselves to DES: each formulation of the PD is carefully constructed so that the players' allowable moves are **constrained**, because it is the relationship of those constraints (and the players' strategies) to the outcome of the game that the modeler wants to understand. In a DES, there are no "strategies" because the entities are reactive and have no agency, and there are no "game rules" because each entity determines its own reactions to external events. A simulation of PD or other game must be constructed differently: it does not make sense to play a game in which the

players dictate the allowable moves and rules. It would not make sense for the agents in a PD simulation to decide on their own, for example, that they want to break out of their prison cell and escape: that is not the relationship that the modeler wants to understand and the simulation would therefore be pointless.

One way to express this is that in an ABM, the agents are in service to the simulation, and not the other way around. The modeler is interested in the relationship between the constraints of the system, the strategies of the agents/players, and the subsequent evolution of the system, and to find that relationship, the constraints must be strictly under the modeler's control. In contrast, a DES is typically reversed: the simulation is in service to the agents. Typically a DES is used by a modeler who is interested in the relationship between the properties of the component parts of the system and the overall system evolution, and to find that relationship, the modeler must be in strict control of the properties of the component parts within the simulation. For example, a circuit may behave one way if one of the parts is a resistor, and another way if that part is instead a capacitor, and it is these differences that the modeler is interested in. If the simulation did not allow him to express "resistance" instead of "capacitance", he would not be able to explore this relationship. The distinction is necessarily a gray one but the implications are non-trivial. We will see evidence of this distinction when we detail our new object model for ABM later. The common parts and language of our ABM object model originate largely in these characterizations.

We now turn to the issue of *locality*. Briefly, our thesis is that the systems that we are interested in have the common trait of exhibiting considerable locality in space/time, and that we can take advantage of this trait in the definition and implementation of our object model. The locality observation is not a new one: some definitions of CAS explicitly incorporate language to this effect, e.g. "a CAS system is one composed of many parts each interacting **locally** with other parts and their environment." [Swarm93] Indeed, one of the most scientifically intriguing (and unintuitive) qualities of CAS, and thus a quality often of primal importance to the analyst and one that must then be representable in the simulation models, is the tendency for seemingly globally organized behavior to spontaneously emerge purely from entities acting locally with only local information, so-called *emergent behavior*. The implication here is that not only can agents act only locally, they also have only local **information**, where local is defined in terms of space-time: any information that comes from distant space must also be distant in time (i.e. old information), or equivalently, there is no instantaneous transmission of information over long distances. From this it follows that agents themselves will generally be restricted to employing fairly localized strategies.

Restricting agents to localized strategies would not be very reasonable unless such agents are both common in the systems being analyzed and relatively competitive in the "game" presented to each agent. Fortunately, we can argue that this is the case. One of the motivating factors for using large-scale simulation to model complex agent-based systems is precisely because they are intractable to simpler modeling methodologies. Because they are intractable to us as modelers, they are also intractable to the agents that are a part of them. The same factors that make it difficult for us to understand and predict

the time evolution of these systems generally make it fruitless for an agent to employ complex long-range prediction and optimization as a behavioral strategy. Stocktraders do not, for instance, predetermine long in advance the day that they will sell a purchased stock, because they cannot realistically predict (in general) the time evolution of the price of the stock. Instead, they make very localized (in time) decisions, e.g. the price has hit a certain goal number so sell now and take our profits and if the stock keeps going up, so be it. This is part of the very nature of the systems to be studied with this modeling technology. Looking at another domain, a pathfinding agent may not be able to see very far ahead; any strategy based on finding a globally optimal path would therefore be pointless, and a local strategy is the best that can be done. We claim therefore that this is both the nature of the underlying systems we are modeling and a competitive strategy for our agents as realized in software, and these observations are reflected throughout our infrastructure definition and implementation.

It is worth noting, of course, that phenomenological systems that do not fit our characterization are likely to suffer from a mismatch with our infrastructure. Such is the nature of simulation: no single technology is appropriate for all applications. We aim only to provide something useful to an important subset of applications.

A New Conceptual Object Model for ABM

Here we define the component parts of our conceptual model. We assume throughout a standard object-oriented software model, including liberal use of Design Patterns. A review of the OO model is beyond the scope of this paper, but we will emphasize certain aspects that we utilize strongly here. We assume a strict model of *encapsulation:* the internal state of an object is not directly visible or modifiable by other objects, and an object's state cannot be modified by another object except through method calls that that object chooses to expose. We make liberal use of the Interface pattern: interfaces are used to define the concepts extant in an ABM, while specific implementations hide details of data storage and algorithm behind their implemented interfaces.

Simulation Objects and Containment

The fundamental unit of our conceptual framework is a *simulation object (SO).* The more general concept of Objects form the basis of object-oriented programming models and are a fundamental concept in the epistemology of human thought and language, a connection noted by, e.g., Booch [], and our notion of an SO builds on that connection. Shortly, a SO is a self-contained collection of state and algorithms for evolving that state, along with methods that define communication protocols by which other objects may influence its internal state. Intuitively, an SO represents anything that exists in the "simulated world", just as the common notion of an object in everyday speech is a representation of anything in the real world. The main concept we add to an object in the OO sense to create SOs is the notion of internal evolution of state over time: some OO objects (those meant to represent things in the real world) have this concept, but others (e.g. objects used purely for software engineering like builders) do not.

It is obvious that SOs might often *contain* (also known as *"has a"* relationships) other SOs, e.g. a car contains tires, or a building contains rooms. However, as opposed to the software engineering definition of containment, we take containment between SOs to refer to containment in the sense of common usage, e.g. physical containment. Thus, we can capture the pattern of containment between SOs as a *tree* (we assume a tree rather than a forest so that there is a single root node discussed shortly): each SO is a node in this tree, and a containing SO is a parent in the tree to its contained SOs.

All trees necessarily have a root, which leads to the following definition:

The root node of the SO tree is that SO in which all other SOs are contained (transitively); we refer to this SO as the *simulated world.*

These definitions and requirements are intuitive and easy to work with. A modeler must define their simulated world and populate it with simulation objects, each of which might in turn be composed of other simulation objects, etc. A common pattern that fits this model is that the simulated world contains portions of the world's domain; each subdomain might contain an environment object which itself contains some collection of passive objects, e.g. rocks or buildings; each domain might also contain active agents; buildings might contain rooms; etc.

One of the less obvious useful implications of this framework is that it forces the modeler to decompose their simulation *hierarchically*, which has several advantages which we discuss below. The existing toolkits have little if anything to say about the decomposition of the simulated world; consequently, the simulation writer is left to discover the advantages of hierarchical decomposition on their own, if at all. Our model influences them to think about a useful hierarchical decomposition from the outset of model development.

One advantage of hierarchical decomposition that we discuss in more detail later is that it maps well to parallel computers. Hierarchical decomposition is a strong form of encapsulation which is known to promote software reuse and composability. Finally, hierarchy and *multiscale* formulations of simulations go hand in hand, and there is considerable evidence from the continuous simulation community, e.g. multigrid algorithms and adaptively refined mesh formulations, that multiscale formulations of simulations have fundamental advantages versus singly scaled formulations. It is natural to construct containment such that containing objects have natural time and space scales that are *larger* than the scales for contained objects, e.g. subdividing a domain into smaller domains, and thus a multiscale formulation often flows naturally from a hierarchical decomposition.

Combining our strong form of containment with strict encapsulation dictates an important consequence. Assume that B is contained in A. Our insistence on strict encapsulation

dictates that A and B cannot "see inside" of each other's implementation, which includes the unintuitive consequence that an object B cannot "see" other SOs contained in A, where "see" is meant in the object oriented sense that it cannot directly access A's interface and thus cannot invoke A's methods. (later we discuss "sight" in the simulation sense and show that B can physically see other agents if the simulation allows it). This generalizes to all forms of interaction:

SOs may interact directly only with the SO that they are contained in or that they contain.

Figure 1 illustrates this concept between SOs A, B and C, where B and C are contained in A. Clearly, if B and C are both contained in A, then B and C have a peer relationship in the simulated world, as illustrated in the first part of Figure 1. However, because of our strict containment requirement, B and C do not have a relationship in terms of object oriented software, as illustrated in the second part of Figure 1: they are both pointed at by A, but they do not point *at each other.* This is a considerable break from most published agent-based modeling frameworks, which often not only allow direct peer communication, but are actually *formulated* in terms of peer communication. However, there is a simple way to reintroduce the notion of peer-peer interaction within our conceptual framework: peer entities such as B and C in Figure 1 can interact *implicitly* through their containing SO. At its simplest, for example, the SO A in Figure 1 can just forward interactions from B to C and vice-versa. However, formulating it in this way allows the designer of the A component to control the processing and transmission of information as a "rule" of the multiplayer game (referring to the discussion of games above), e.g., A can determine if a bullet fired by B actually hits C instead of allowing B to dictate the result. Thus we see that our model subsumes the more common model as a special case, but allows more control for the modeler.

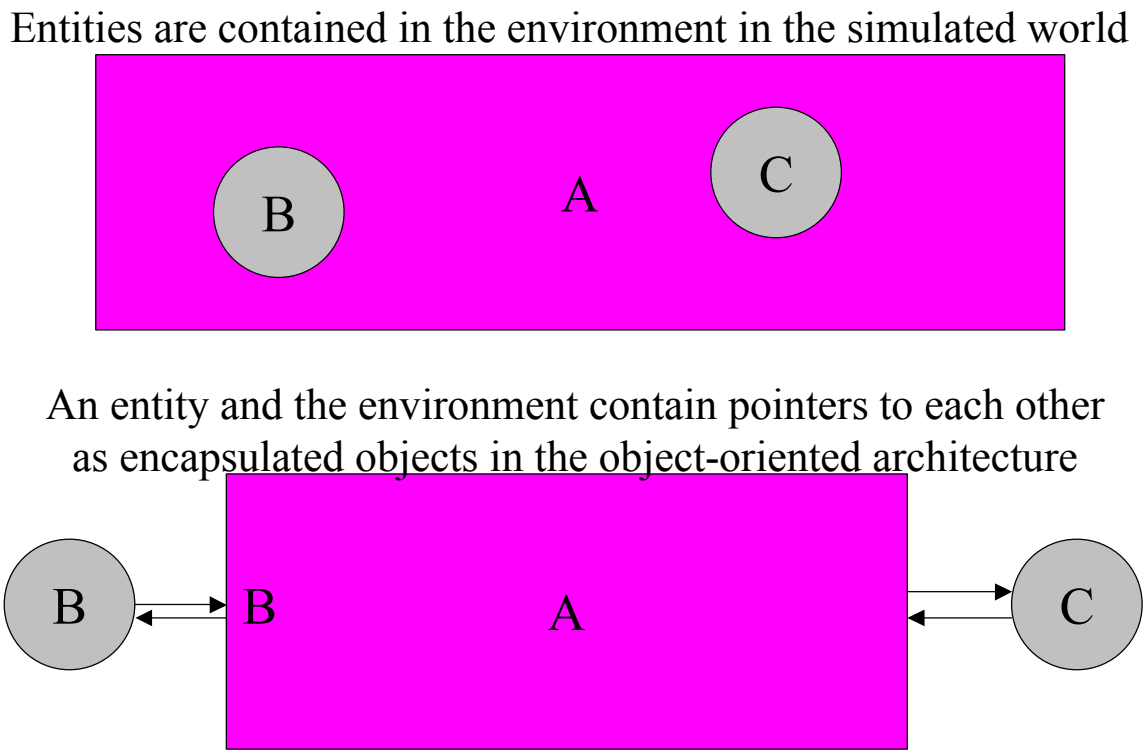


Figure 1: An illustration of two separate but often conflated relationships between simulation objects and their environment. The top figure shows that in the simulated world, SOs B and C are physically contained in the environment A. The bottom figure illustrating the object-oriented architecture illustrates that while references to B and C are a part of A's implementation and vice-versa, because A is fully encapsulated, neither B nor C can see the rest of the internal implementation of A, *including the fact that other agents are part of A's implementation.* Because of this, B and C cannot directly interact with each other, but must instead interact through A.

We now introduce two key, paired subclasses of Simulation Object: *agents* and *environments.*

Agents

Our model represents *agents* simply as a special class (and thus a subclass in OO implementation) of a SO as it is a strong form of encapsulation. The motivation behind agents is that they are people or other members of a population that have *agency,* that is, they make decisions to modify their internal state for the purposes of optimizing some internal objective function as achieved and measured relative to the simulated world (another commonly used term is that Agents are the *actors* in the simulation because they are the objects that take *actions*).

The definition of Agent is highly context specific and is part of the art of modeling. For example, in most simulations, a plant would be a SO but not an agent; in a simulation of the dynamics of forest growth, however, the tendency of a plant to grow towards sunlight (that is, the plant "chooses" which direction to grow) and to reproduce may dictate representation of the plant as an Agent.

In terms of design patterns, we use the interface Agent mostly as a *marker interface* (it probably makes sense to add a small number of bookkeeping methods to this interface, i.e. Get/SetEnvironment, so arguably Agent is not literally a marker interface). In a given simulation, agents will have additional methods that they must implement to be agents rather than SOs, but these are generally simulation specific. We do however specify certain relationships between SOs that implement the Agent interface and other objects, as we now discuss.

Environment

Existing models for agent-based simulation are generally described as including agents and their *environment,* and we retain this notion very explicitly:

An *environment* is a SO that allows contained agents to interact with it.

We consider this an *iff* relationship: if an object allows contained agents to interact with it (directly and not transitively), it must be an environment. Since environment is a specialization of an SO, the obvious OO representation is that environment is an interface that inherits from SO. Shortly, an Environment exposes additional methods that define the means by which Agents can interact with (and thereby influence or modify) the Environment object to contained agents.

We mentioned in our discussion of Agents that specific simulations may require Agents to implement additional methods. This manifests in the following form:

Subclasses of Environment may require that their contained agents implement one or more subinterfaces of Agent, and subclasses of Agent may require that their containing Environment implement one or more subinterfaces of Environment.

Note that this is a slightly unusual pattern in OO software. Typically, OO software is thought of as "client/server": as long as a server implements a certain interface, any client may use it. Here, however, we are requiring a pairwise "match": an Environment and an Agent may only interact if they match each other's requirements. Not all Agents can operate in all Environments, and not all Environments can hold and service all agents.

This is the mechanism by which agents are matched to environments. It is also a means by which we can approach reusability of agents and environments: agents and environments are matched by this pairing of subinterfaces, rather than the more restrictive pairing of a specific agent class with a specific environment class. Agents that implement certain agent interfaces can be inserted into any matching environment, even if that was not the intended environment, at least from the software engineering point of view. By this, we refer to the following fundamental notion of interoperability: if it makes sense for agent A to take part in a simulation defined in environment B from a

modeling point of view, the software architecture should not block that from happening; if it does not make sense, no software can alter the semantics to force it to make sense. E.g., if a pathfinding agent A is inserted into a new environment B in which it makes sense to pathfind, hopefully the software architecture will make insertion of A into B straightforward; however, a pathfinding agent inserted into, say, a social simulation and environment does not make semantic sense, and thus a software architecture that does not allow insertion of this agent into that simulation is appropriate.

Multiple Inheritance

It is important to note that we do not assume that Environment and Agent are mutually exclusive: a single object may simultaneously be both. Fortunately, this maps naturally into our proposed implementation through multiple inheritance of interfaces: objects of a class that multiply implements both interfaces have both roles in the simulation. In this way, an Agent in one Environment can itself be an Environment for other Agents, a capability that enables the construction of complex hierarchical models, whose benefits we detailed in the introduction.

Domains and Relationships

It is widely agreed that a key driver of the dynamics of a complex system is the *relationships* and *interactions* of its component agents, and yet our model explicitly forbids the agents from interacting directly with each other, another seeming paradox. To illustrate our solution, we refer again to an example. A car (certainly a complex system) has a steering wheel and it has tires, and the dynamics of the car are critically dependent on the relationships between the tires and the steering wheel (as well as other parts). Some cars have manual steering and some have power steering (and I have now exhausted my automotive knowledge) for example, which results in a different dynamical response between turning the wheel and the action of the tires, even if the steering wheel and the tires are exactly the same in both cases. What this indicates is that the relationship between the tires and the steering wheel is not a property of either component part; it is instead a property of the **car**. The distance between the steering wheel and the tires, the intervening parts (manual vs. power steering), etc., are all part of the internal state of the car. Our epistemology generalizes this example:

The relationships between agents contained in the same environment are a state property of the environment, not of the agents.

This is somewhat unintuitive, as we can see from any example that includes spatial locality: *position is a state of the containing agent, not of the contained agent.* Consider a person walking down a street: what is their position? Traditional simulation epistemologies do not give guidance to this question, and as a consequence, it is common to find position implemented as a state variable of the **person**. And yet, position is clearly context-dependent. In a simulation of the city in which the person is walking, their position might be the name of the street they are on. In a simulation of the entire globe, it might be their latitude and longitude. In a simulation of a battlefield, it might be a grid

location used to represent the battlefield. Just as the inclusion and position of the tires on a car are a state property of the car, the inclusion and position of a person in some geographical environment object (for example) are an internal state property of the geographical object.

We illustrate with the property "position" because it is instructive in considering *relationships*. Simply put, *the position of agents B and C within environment A is not only an internal property state of A, but it also defines a relationship between B and C in the context of A*. Again, this is easy to illustrate with an example: if B is a soldier interested in shooting C, clearly their relative positions define (possibly with other additional state internal to A) the nature of the "can I shoot you?" relationship. If their positions are too far apart, the induced relationship could be "no, you are too far away to hit the target"; if they are close in distance but A's internal state includes some obstacle between B and C, again the answer would be "no". Recent discussions on the Repast mailing list on the next generation of Repast grid classes reflects a similar theme, where various researchers have tried to generalize the Repast grid classes based on the observation that their purpose in an EBM is merely to track spatial relationships between agents. For example, there has been discussion of the notion of generalizing grids using concepts such as *metric spaces* or *graphs*, each of which captures the notion of a collection of points (each representing an agent in the simulation) in an algebraic space that can represent dynamically changing relationships between those points. We adopt this notion of relationships as a part of our epistemology without being more specific about its semantics:

The job of tracking relationships between agents is the environment's, not the agents themselves.

Interaction Between Agents

We can continue to use the previous example to illustrate interaction between peer agents (that is, agents contained in the same environment). Our model does not allow them to interact directly, and yet clearly modeling a complex system requires interaction of peer agents (B shooting at and potentially harming C in this example). The example brings up a canonical problem in existing epistemological simulation models: which component of a model in environment A determines the effect on a simulation object C of an intended interaction initiated by simulation object B? In our example, this could be: which component, A, B, or C, determines whether or not a fired bullet from B actually hits C? Clearly, there is a fundamental problem with allowing the agents themselves to determine this: B and C could come to different conclusions, resulting in an unrealizable simulated world in which B treats C as dead and C treats itself as alive. This type of example hints that what is needed is a neutral third "party" to determine the resolution of agent interactions. Our epistemology wraps this into a conceptually clean package:

The enclosing environment is responsible for moderating interactions between its contained agents.

In our shooting example, our conceptual framework requires that the modeler models interaction between B and C as a pair of interactions: between B and A, and then A and C. Everything must "go through" A, not only conceptually, but literally in software. Referring again to Figure 1, the bottom picture shows clearly that since B and C do not have a direct connection, they must interact through A.

There are of course different ways to model our example within this epistemological framework, but a straightforward way is to introduce the notion of the bullet as a separate object. Here, the interaction between the shooter B and the environment A has the form "A expels a contained object D in a certain direction and with a certain velocity." A contains B, B contains "a bullet" D, and since B controls its internal state, it can choose to expel that bullet, albeit according to the constraints of its own internal dynamics (which in this case includes a gun, or else it is highly unlikely that the presumably thrown bullet is going to travel very far or do very much damage). Once the bullet has been expelled into the containing agent A, its position within A is an internal state property of A. Indeed, were A a model of an ocean for example, the flight of the bullet would be significantly different than were A an open field, and indeed that is the point of encapsulating all environmental information into a single encapsulated object: the "physics" of the simulation (equivalently, the rules of the game) can be changed without modification to the other components of the simulation, particularly the agents. From this point, it is up to A to determine the relationship and thus possible interaction of the bullet with other agents contained in A. If, as B intended, A determines that the bullet eventually changes its positional relationship to C to "zero distance" and that therefore an interaction between the bullet and C is required as part of the evolution of A's internal state, A can then initiate an interaction between the bullet and C. At this point, A informs C that an interaction is occurring along with the parameters of that interaction [as discussed before, the straightforward way to implement this in OO software is as method calls that A makes on C], and C updates its internal state to reflect the fact that it has been shot. Note that how C updates its internal state is entirely encapsulated internal to C, as per our encapsulation requirement. This makes sense, as only C can know the details of its internal structure (material properties, etc) needed to properly account for the effect of this interaction on its internal state.

Forces and "Push" Models

The choice of *push* versus *pull* simulation models is one that has been around since the beginning of simulations. We explain the difference with a detailed discussion and show that our choice is explicitly a **push** model:

In our example in which an agent B attempts to shoot another agent C interior to environment A, we detailed that our epistemology requires this to evolve as a pair of interactions, one initiated by B with A and a second initiated by A with C. As we will see in the sequel, B's initiation of the gunshot is called an "action" in our framework. Our framework must also have a conceptual place for A's initiation of an interaction with C, the target of the gunshot. Just as A must expose actions that may affect its internal state, contained agents like C must expose the means by which its internal state may be

influenced externally. Again we adopt the mechanistic model that any influence on C must be the result of a change of internal state of A. There is a necessary asymmetry in our model in that contained agents possess agency, whereas an environment does not. For this reason, we introduce the concept of a *force.*

We define a* force *to be anything external to a SO that might cause an internal change of state within that SO.

In our shooting example, an agent incorporated into a militaristic simulation would reasonably be required to expose methods by which the environment can inform the agent that it has been shot. "Being shot" is a high-level language that is shorthand for "an object with the velocity, mass, and other properties generally associated with a bullet has just delivered a corresponding "force" to you". Just as the language of actions can be shorthand abstractions representing a collection of smaller mechanisms, the language of forces can also be that of a shorthand abstraction. The actual choice of language is up to the modeler of the particular simulation; our epistemology clearly delineates the roles and responsibilities of the environment and agents, but not the particular language. A force in our epistemology can be any mechanism that can impinge upon an agent, and is highly context dependent. Certainly, physical forces are covered, but more abstract interactions are also included. For example, in a social simulation, a "force" might be a phone call initiated by another person; we also explain below that *perception* is modeled as a force in our system.

Our resulting model of interaction is a *push* model by design. By this, we mean that interaction is initiated by the producer of a force and then "pushed" to a contained or containing SO. There are several design considerations that went into choosing this push model of interaction. A push model promotes a "cause and effect" formulation of the modeler's system, and this maps well into several aspects of our proposed architecture. Cause and effect maps well to parallel computing because it translates to "one way" communication between simulation objects in which a cause at simulation time T1 creates an effect at simulation time T2; if T1 and T2 are on separate processors, the delay between T2 and T1 potentially gives the parallel implementation time during which information can be transmitted between the two processors (*look-ahead* in DES parlance), and since communication delays are the single largest cause of lack of performance in parallel programs, this offers considerable promise for good parallel performance. A "pull" model in contrast would require two-way communication, creating a potentially costly synchronization between every SO interaction.

Less obviously, the push model promotes composability because it is an *explicit* time formulation: what happens at one point of time is purely a function of previous times instead of also being a function of the current time. In traditional physics simulations, the tradeoffs between explicit and implicit formulations are well known: explicit formulations are more flexible and additive, but are generally less accurate. We have more to say about accuracy of ABMs later, but we preview that with our claim that because most ABM formulations introduce error and approximation in much larger

proportions than physics calculations, it follows that accuracy requirements during the calculation are much less in ABM simulations than in physics simulations.

Actions

As we have seen, the model we have discussed to this point specifies that agents can interact with their environment. Here we discuss the epistemology of interactions initiated by the contained agent, which we define as *actions.*

To motivate our model, consider a bird in flight. The bird is an agent contained in an environment, and the bird's flight is reflected as a change in the internal state of the environment since the position of an agent B contained in environment A is a property of A's state, not B's. Typically, existing epistemologies would allow the bird to "change its position", from which it follows that movement in either a straight line or a change of direction would both be indicated by the bird as a change of position. This cannot be the case in our epistemology since position is a function of the environment, not the bird. Instead, we can draw inspiration from one of Newton's laws of motion: an object in motion stays in motion until acted on by an outside force. Our inspiration here is that the bird continues to fly in the current direction until the sum of external forces acting on it changes, so that in particular, the bird flying in a straight direction does not need to "change its position" periodically; instead, the environment will update its internal state periodically to reflect the bird's continued straight flight. Those forces could change as a result of a change of state in the environment, e.g. a change in wind direction, but here we consider a situation in which the bird chooses to change its direction of flight and how it fits into our epistemology. An agent can change only its own **internal** state, and yet the bird's direction of flight will only change if **external** forces acting on the bird change. The solution is to recognize that the forces acting on a bird in flight are a function of the relationship between the bird's body interacting with the air, and that a bird changes its direction of flight by adjusting the position of its body (its wings, etc) so that the resultant force vector induced by its relationship with the air changes. Thus, the epistemological principle is that *agents can only influence the state of their containing agent by adjusting their internal state.*

For this principle to translate into a simulation context requires that the enclosing agent A recognize that B's internal state has changed; for our example, the environment must know that the bird has changed its wing configuration. Since B's state is encapsulated and thus hidden from A, B must explicitly notify A that is has changed its state. In a simulation realized in software, the straightforward way to implement this is to have B make a method call on A indicating the change of internal state and the parameters describing that change.

At first glance, this model appears awkward and overly detailed. In our example, it would be unnecessarily complex to implement the bird's change in flight by implementing the physics and language of aerodynamics and biomechanics just to implement a simple change of direction by the bird. However, our framework says nothing about the level of

accuracy or detail required; instead, its point is to clarify the split of responsibilities between the environment and contained agents. The actual choice of language for informing A of B's change of internal state is up to the modeler. Macro-level language is not only allowable but is envisioned, e.g. a method call to the extent of "I've changed my internal state in a way intended to make me turn left" rather than "I have tipped my wings and reshaped them into a slightly different parabolic arc with the following properties". In other words, the change of internal state does not have to be communicated in the language of B's internal state; it can also be expressed in a shorthand form in terms of the intended change to A's state. The concept of an *action* captures this notion. *An action is a change of internal state by a contained agent that has the potential to change the time evolution of the internal state of the enclosing environment.* Actions are the means by which contained agents initiate interaction with their enclosing environment and as such are the means by which agents attempt to meet their internal objectives. In terms of agency, actions can be seen as the "moves" that an agent makes to try to achieve their internal objections. In an OO software implementation, the straightforward mapping is that the containing agent A exposes certain methods as allowable actions to its contained agents, and contained agents make calls to these methods as their internal agency dictates. Implicit in this model is that any potential actions not exposed as methods by A are assumed to have no measurable effect on A's internal state: A dictates what can and cannot effect its internal state, which is entirely consistent with the encapsulation principles of our model.

Push/action model enables reusable environment implementation:

Strict adherence to the push model guarantees correct execution (without expensive rollbacks) of a simple reusable algorithm for implementation of ABM environments. The algorithm runs both sequentially and in parallel equally well. This allows us to create generic implementations of an ABM environment that can be reused in a variety of ways to greatly simplify the task of creating new ABM environments.

The basic structure of the environment's main loop follows:

```
Initialize Schedule with each agent's initial actions and
other events.

Do until simulation ends:

  Pop next event E at time T off of the Schedule

  For each agent touched by a force resulting from E and
for the agent whose action scheduled E:

    Advance the agent to time T, allowing the agent to
initiate new actions. Each new action's resolution must
occur after T.

End Do.
```

The correctness of this algorithm depends intimately on the "push" model and the concept of actions. The intuition here is that agents generally perform some action for some period of time and are willing to "contract" with the environment that *unless something interrupts that action*, they will continue to do that action for as long as they say they will. For instance, in a path-finding simulation, the agent may be willing to contract that it will "walk east for 5 minutes unless I bump into an obstacle or something else major happens to interrupt that walking." This is a natural way to program agents; in addition, it provides the generic environment implementation above. With this information from each agent, the environment can calculate the next simulation time at which anything can happen that can affect any agents, and more importantly, it can determine that unaffected agents do not need to have their internal state updated. To show the importance of this from an execution efficiency standpoint, consider the alternative, generally known as **time-stepping:** in this algorithm, the states of all agents and other simulation objects are updated *whenever any event occurs* (this is called time-stepping because often this is implemented by having a generic "event" at regular time intervals whose only purpose is to cause the updating of the internal states of all objects). The time-stepping algorithm is safe in that it assures that no calculation is done using stale state from some SO, but it is a massive overhead in that it requires visiting every agent and object at every event time.

Time-stepping can still work if the simulation writer only uses fixed length event scheduling, but that is necessarily somewhat restrictive and is in particular a roadblock to composability in that components must have agreed on the interval of such regular scheduling ahead of time. A component written for 3-minute intervals, for instance, will not work well in a simulation in which 5-minute event intervals are taken.

Our generic algorithm provides considerably more flexibility, however. Only the agents actually affected by an event are visited by the environment. Time-stepping is easily recovered should that be desired, e.g., the simulation writer can create an event on a created object whose only function is to cycle over all of the agents and visit them, but it is much more general than time-stepping and in particular is efficient when mixing and matching components from different developers. At any given time of an event, some or perhaps many of the simulation objects and agents will exist in an "old" state, but because of the semantics of the push model and actions, correctness is preserved: an "action" is literally an immediate change of internal state. When an agent states that it is going to participate in an action for a certain period of time unless interrupted, it is stating that in terms of its effect on the world outside of it, its state will not change for that time. Literally, then, as far as the rest of the simulation is concerned, that agent's "old state" is still valid. In effect, we are taking advantage of the *discrete* nature of discrete simulation to create a more efficient simulation engine. Note that since it is the simulation environment that delivers forces to each agent, the simulation agent can determine whether or not the agent will be interrupted before the end of its current action, meaning again that it can guarantee correctness on its own.

The last part of this generic algorithm is worth a brief discussion. If the simulation environment delivers a force at time TW to an agent that is at some past time T1, according to our algorithm, the agent must advance its internal state from T1 to T2. During this processing, it is possible that the agent may produce a force that will be expelled into the containing environment at some time less than T2. This seemingly creates a dilemma: the environment has advanced its state to T2, most likely advancing the states of other agents and SOs, so that if the release of a force from the agent at some time before T2 affects the environment or other agents at some time before T2, many states will be incorrect. While some simulation frameworks address such issues with complicated techniques such as time rollbacks, we adopt a much simpler semantic solution: this is part of the "contract" that agents have with their environment, and the contract simply prohibits this eventuality. If an agent commits to an action until some time T2, then its part of the bargain is that it agrese that nothing that it does will cause an effect on the environment or another SO until at least time T2.

Clearly, the utility of this semantic contract depends on its implications in practice. Fortunately, we argue that this is not only restrictive, it is a natural extension of the ideas of time-stepping. Time-stepping clearly includes this notion but in a more regimented fashion: no agent may undertake an action that has an effect that takes place less than the length of the time-step in the future. Time-stepping effectively discretizes time and in the process creates a "minimal reaction time". Our model incorporates the notion of a minimum reaction time, but generalizes it so that it is not uniform across agents and across effects and forces. A statement by an agent that it will perform an action for X time unless interrupted is equivalent to a statement that "if I am interrupted, it will take me at least X units to react", and we can use this reaction time to create a more efficient simulation engine. We allow more of a negotiation between the environment and the agent than time-stepping, which requires a uniform minimum reaction time as dictated by the environment, but both are based on the same notion: if there is a reaction delay between causes and effects, the state of the simulation is constant during that time and needs not be updated until the next "effect" time.

Perception

Arguably, one of the issues in which existing frameworks allow too much room for error is *perception*, or in particular, the difference between perception of the simulated world and the simulated world itself. An agent making its way in a simulated world must have a way of perceiving that world, and yet our framework has insisted that an agent cannot access the internal details of the environment. It follows that perception must occur as inquiries to the environment. One of the conceptually rewarding aspects of our model of actions and forces is that perception does not require additional concepts: actions and forces are sufficient.

We claim that quite literally, sensing one's environment is achieved mechanistically by the arrival of forces on the body of the agent. Sight, for example, is caused by light impinging on the eyes. Sound is caused by vibrations of air impinging on the ears. Smell is caused by the arrival of minute particles at the nose. Etc. Thus, the act of perceiving by

the senses fits into our epistemology as a set of forces arriving at the agent, and can be implemented just like any other force as parameterized method calls that the agent exposes to the environment.

For the force model of perception to fit into our framework, we must explain how the arrival of a "perception force" may change the internal state of the agent, just as the arrival of a bullet may change (will change unless, say, wearing a Kevlar vest) the internal state of the agent. In other words, what internal state changes when the external world is perceived? The answer is that the agent's *mental model* of the world changes. The concept of a mental model is hardly new, being strewn about the literature of several fields, including artificial intelligence on the computing side and the study of consciousness on the philosophical side. The concept is that an agent does not directly know the world; instead, agents build an internal representation of the world based on the perception data that they receive. In general, that internal representation can span a spectrum from the fairly detailed kind of mental model that humans or sophisticated AI agents might build, down to the incredibly crude such as the plant growing toward the light we discussed earlier: a plant does not even possess a nervous system in the sense that we usually mean it and its "mental model" is probably physically encoded as a simple set of chemical gradients or some other very simple process, and yet a mental model it is (and now I've exhausted my botanical knowledge as well as my automotive). In either case, a mental model is definitely an internal property of an agent, and new perceptions (such as seeing a speeding train hurtling towards you that you had not noticed before) may change your mental model and thus your internal state (and, in keeping with our epistemology, a change in one's *actions* would presumably then follow).

As described so far, the arrival of perception forces at an agent has been passive on the part of the agent. In a software realization, if perception is always passive, it would require an overwhelming number of method calls from the environment to the agents to continually keep them informed of the current set of perception forces incident on them. Fortunately, our model's inclusion of actions solves this dilemma nicely. Specific details are dependent on the specific simulation and the choices that the modeler makes, but we can illustrate with some examples. One possibility is to include a form of active perception as an allowable action to the agents, e.g. "look around". Here, the agent initiates an action with the environment of "look around", and the environment responds by making "arrival of perception forces" calls on the agent. In terms of our epistemology, the underlying mechanism is along the lines of the agent informing the environment that it has changed its internal state so that it is now receptive to a particular type of perception force, or equivalently, the lack of a perception action is a note to the environment that perception forces will not currently cause a change in internal state in the agent. An example with a different intended effect is that of an agent watching for some particular thing to occur, e.g. for a person to enter a room. An elegant way to fit this into our model is to have the agent call a method to the effect of "I'm watching for someone walking through that door," and for the environment to implement this as a type of "triggered event". Here, no method call indicating the arrival of a perception force would be made by the environment on the agent until and if the triggered event occurred, that is, until a person walks through the door in the simulated environment. At that point,

the agent is informed that it has perceived the event it was looking for, it changes its internal state by changing its mental model of the world to reflect the new entry, and it may then choose to initiate new actions in response to its new mental model.

Time

Existing EBM conceptual frameworks borrow heavily from discrete event simulation, and as such time is very much a central part of the model, particularly the concept of *events.* To this point, we have made only tangential reference to time, and its place in our epistemology has not been detailed.

One place for time in our epistemology is as an internal state of objects, most notably but not exclusively an internal state of the environment object. The other place that time occurs in our conceptual model is as parameters to actions and forces. Waxing philosophic for a brief interlude, one way to view time is as a "fourth dimension" akin to the three dimensions of space. We have detailed above how space's role in our epistemology is as an internal mechanism for an enclosing environment to track relationships and interactions between objects in that environment, and the notion translates to the fourth dimension of time as well: time is a form of relationship between objects in the simulated world along a fourth dimension. In this view, it is consistent that time is an internal property of the environment used to track relationships and interactions in the same sense that space is used. Indeed, the analogy holds in other ways, for example, we illustrated the contextual/relative nature of space, and the same holds true for time: the units of time, the origin of the time axis, etc., are all dependent on the context of the environment. A simulation of the climate may step in 1,000 year increments and begin at the formation of the earth; a simulation of a battlefield may step in seconds and begin and end in 10 elapsed minutes; a simulation of an economic game may proceed in "turns" that do not even have an explicit representation in our usual time units; etc.

In a recursive situation (discussed in more detail below), it follows that the time of an enclosing agent A will in general be different than an enclosed agent B, but this is not the paradox it appears. Assume that A is a person in a city, and B is a cell in A. Just as the appropriate spatial representation of the city is different than the appropriate spatial representation of the cells inside of a body, the same time representation would not be appropriate, not only in units but also in origin. The person's origin might have been their birth tens of years before, while a particular cell may have come into existence much more recently than that. There is no need to normalize the cell's timeline to set "tens of years ago" as the origin. To put it succinctly: time is relative.

Actions and forces may have a time parameter, allowing a bridge between the environment and the agent regarding time. Like all action and force methods, the specific form is up to the modeler and the specific simulation, but examples can illustrate some of the choices. Actions may have a *duration* parameter, for example, "I will walk in a straight line for 5." Durations do not have to be expressed in normal time units, e.g. "I will walk in a straight line for the minimal amount of time." Here, the environment

determines the "minimal amount of time" according to its own implementation: this kind of language for expressing durations is handy for decoupling an agent's sense of time from the environment's sense of time for the purposes of being able to dynamically change the "time discretization" of the simulation. Likewise, forces may have time parameters: "you will be bombarded by gamma radiation for the next 5 minutes." Duration parameters can also be contextual, e.g. "I will walk in a straight line until I bump into something." Often, these can be combined for best simulation effect: "I will walk in a straight line for 5 minutes or until I bump into something" prevents the realizable but somewhat farcical situation in which an agent continues walking even though blocked by a barrier. A general principle of our motivation behind the development of our conceptual model is illustrated by this example: the model is intended to create *realizable* simulations, but stops short of attempting to prevent *stupid* simulations, as the latter is too subjective to encode.

Although our epistemology does not require this, a common implementation of an environment would be in terms of a discrete event simulation, as it is a convenient representation of a time-evolving system. Thus, "I will walk for 5 minutes" might be implemented by the environment object by placing an event on an event queue 5 minutes from the current time representing the arrival of the agent at a new location in the environment, and this is the implementation that our generic environment described above uses. This connection is by design: *a way to look at our epistemology is that it creates a higher-level abstraction layer that sits on top of a discrete event simulation.* In the context of our previous statement, one of the potential pitfalls of discrete event simulation is that as a low-level programming layer, it is easier to create unrealizable simulations. By putting a layer between the modeler and the discrete event simulation, we hope to improve the modeler's lot.

Constructing Environments: Rulemakers, Mediators, and Domains

Here we suggest a standard set of objects and responsibilities that are internal to Environment SOs. This represents an additional layer to our design which arguably is not required for the rest of the design to provide value but increases that value. The three components of this subdesign are Rulemakers, Moderators, and Domains, which we now discuss.

A ***RuleMaker*** is the component of software (an object but not a SimulationObject) within an Environment that exposes methods to SOs, particularly Agents, by which the SOs might influence the internal state of the Environment (and thus implicitly other SOs and Agents contained within that Environment). These methods define the forces that can effect the Environment. Internal state changes may include immediate changes to objects within the Environment, but generally most of the internal state changes are to the Environment's event queue, representing the possible arrival of forces at other Agents and SOs.

A ***Mediator*** is the component of software within an Environment that has the converse responsibility of the RuleMaker: it pushes effects out to Agents and other SOs in the form

of forces. A RuleMaker implements its methods representing actions by agents and other forces impinging on the environment by scheduling future events with a Mediator object. When called, that Mediator object calculates the effects of the force impinging on the Environment as a set of new forces impinging on other objects, and makes method calls to those SOs informing them that they have been affected by something external to them. It is the mediator's job to determine what happens to the force between the originating SO and the target SO, as in the case of our example of a shot bullet from above.

A Mediator must interact with the ***Domain*** object to make this determination. A Domain represents the "physical" part of the environment, and encodes whatever physics that exists in the Environment. The Mediator and the RuleMaker query the Domain to determine physical parameters necessary to make their determination. Note that the Domain can also be a SO: if the Domain's state can be affected by actions in the simulation, then it is a SO and must expose methods that the Mediator can use to report the impingement of a force. In some models, the Domain may be immutable, in which case there is no need to represent it as an SO.

Constructing Agents: Effectors, Actuators, and DecisionMakers

Here we suggest several objects and their responsibilities that make an effective standard for implementation of Agents. Their incorporation is not necessary for the rest of our epistemology, but we claim that these functionalities will exist within all agents and thus representing them with these objects provides some hope of reusability. It is also the case that this model of Agents is particularly compatible with the design for Environments given above. These components are Effectors, Actuators, and DecisionMakers.

We can explain Effectors and Actuators succinctly in terms of our epistemology of interaction: Effectors are the objects within an Agent responsible for responding to forces pushed at the Agent by its Environment, while Actuators have the opposite responsibility, that is, they push forces from the Agent at the Environment. Examples of Effectors might be various sensing mechanisms, e.g. touch and eyesight, as well as "response" mechanisms, e.g. the collision of an object with the Agent may require internal state changes (e.g. internal damage), and it is the responsibility of the Effector to implement methods that effect these state changes. Actuators have responsibility for communication in the opposite direction: if the Agent's internal state is computed to create the push of a force out toward the environment, it is the Effector's responsibility to report this in the proper form to the Environment.

There is an obvious and intended relationship between the components of an Agent described here and the components of an Environment described in the previous section: Actuators match up with RuleMakers and Effectors match up with Mediators. A Mediator has the Environment's responsibility for communication from an Environment to an Agent while a Effector has responsibility for communication from an Environment to an Agent within the Agent. Thus, it is required that they "speak the same language" (and the same holds for the other pair of components). It is via this mechanism that we promote not only composability of Agents and Environments, but also reuse of the

component parts of Agents and Environments. Though it is not required, one can imagine taking an existing Environment and Agent, adding to each a compatible pair of either Mediator/Effector or RuleMaker/Actuator, and thus trivially composing a new simulation. This is particularly feasible for "reactive" type processes, that is, those not requiring agent intelligence.

An example in our own work is that of adding disease dynamics to an existing simulation: our model of disease transfer is that of *pathogen exchange* between Agents and their Environment. This can be "glued" onto existing simulations without affecting the rest of the simulation simply by adding a "pushOutPathogen" Actuator and a "recievePathogen" Effector to the Agent, along with state tracking the internal state of the pathogen, via the Composition pattern (and a similar RuleMaker and Mediator to the Environment). The resulting simulation is somewhat simple in the sense that without additional logic, there is no explicit coupling between the behavior of the Agent and their "sickness" state, but there are cases in which this is the desired simulation, and in either case, even if additional logic is desired, the entire implementation is simplified through this reuse. There is no compelling reason to think that the logic of, in this case, pathogen exhange, needs to be implemented anew for each and every class of Agent, and this architectural design enables this kind of reuse.

Composability of Environments and Agents follows from this. Say for example that we want to add an Agent with various Actuators and Effectors to an Environment that requires the pathogen exchange Actuators and Effectors described above, but the Agent does not have them. Through the composition pattern, we can add these Actuators and Effectors to the Agent and enable its inclusion in the Environment. Granted, the new Agent will follow a "standard template" for pathogen exchange instead of introducing its own algorithm, but in many cases this may be sufficient, for example, the modeler may be interested in the coupling between the decision making behavior encapsulated in this class of Agent in the context of the standard pathogen exchange model.

A ***DecisionMaker*** is an object within an agent that encapsulates the "intelligence" of the Agent. It may often be the case that we want to reuse certain decision making algorithms in a variety of Agents; again, the notion here is that the modeler is interested in the relationship between the system behavior and some other aspect of the system than the decision making algorithm, and being able to easily reuse the decision making component allows them to hold that part of the simulation fixed while the desired part varies. A canonical example is that of a *Pathfinding* algorithm. Pathfinding algorithms are hard to write (they are the subject of an entire subfield of AI) and thus it would be frustrating to have to write a new one for each variant of Agent in one's model. By encapsulating this algorithm in a single class, reusability is promoted, though hardly guaranteed. A DecisionMaker can be seen as a bridge between Effectors and Actuators: Effectors take input from the environment and translate it into changes to the internal state, while Actuators take internal state and translate it into effects on the Environment. A DecisionMaker bridges the changes of internal state caused by Effectors into changes in internal state that trigger Actuators. E.g., the Effector records the sighting of an imminent obstacle as a change to the internal state of the Agent's internal mental model,

the pathfinding DecisionMaker reacts that that new internal state by changing the internal state that determines "direction of movement", and that change of internal state is reported by the Actuator to the environment in terms of the Environment's RuleMaker method for reporting a change of direction.

Hierarchical ABMs and Implementation via Hierarchical DES

We have laid the groundwork for hierarchical ABMs above. There are a large variety of variants for how this can play out, but the key observation is that our strict containment policy, which creates a tree of SOs, leads naturally to a hierarchical formulation of the simulation. Given a tree of SOs, the choice for converting it into a hierarchical simulation is basically: which SOs should also be environments? Consider, for illustration, a simulation in which agents exist inside of rooms, those rooms existing inside of a building. The simulation designer can choose to have the buildings be environments, so that agents choose to move within rooms or from room to room; or he can choose to model both the building and the rooms as separate but nested environments. In the latter case, agents move within a room by negotiation with the room environment, but there are two different ways to implement movement between rooms: agents can be expelled from one room and thus into the building environment and then negotiate with the building environment to move to another room; or, rooms could be the agents in the building environment, and they could exchange agents by negotiation with the building environment. There is no single "correct" answer.

Likewise, domains can be directly hierarchical, e.g. in the first implementation, the building environment would have a single, hierarchical domain that includes many rooms, or they can be indirectly hierarchical via nested Environments, as in our last example where the Domain of the building is simply a spatial representation of the location of the various Room agents, but the actual room domain is an encapsulated part of those room agents (and since it is encapsulated, may in general be represented in different ways in different Room Agents).

Details are beyond the scope of this paper, but here we indicate in a high-level fashion the relationship between hierarchical ABMs and our hierarchical DES layer detailed in our companion report. We have noted before that "time is relative", and that in particular, it is a separate state variable for each Environment. Yet, here we have discussed the possibility of nested Environments. Since our canonical Environment implementation includes a DES engine, it follows that nested Environments require nested DES engines. Indeed, that is exactly what we have developed in our companion paper: hierarchical DES engines. The semantics and APIs of our hierarchical DES engines have been explicitly formulated to accommodate the semantics of our push/action conceptual model of interaction between SOs and Environments. In the case that an SO is actually an Environment to other SOs, the semantics of our nested Schedules is that the parent Schedule advances to the first event Time, and then causes all children Schedules to catch up to that time, during which processing they can only schedule events on the parent schedule after the parent schedule's current time. The match between this behavior and the action/reaction model detailed above is one-to-one: the Environment advances to the

next event time, and tells affected SOs to advance to that time as well, though by the contract of reaction time, they cannot make any causal changes to the parent Environment until after the Environment's current time. If the affected SOs are themselves Environments, then it suffices to have the Schedules for the sub Environments be nested with the schedule of the main Environment.

Likewise, the distributed object model developed in our companion paper was developed with the idea of directly serving as an implementation layer for the ABM conceptual model developed here. Objects are always contained within other objects in our distributed object model, representing the strict containment model detailed here. Objects may only move from one object to another object in our distributed object model, thus providing a layer that is convenient for implementing, e.g., the push model for ABMs whereby forces and other objects are pushed from one object into the containing object. Also, the object model detailed in our companion paper is explicitly hierarchical for efficient implementation on parallel processors, matching our claim here that constructing ABMs in a hierarchical manner provides efficient implementation on parallel computers.

Current status of software development:

Parts of the DES layer were implemented in Java using MPI Java and classes from Repast and used to implement a model of infectious disease spread. A skeleton of the ABM layer was implemented in Java. The project to develop both of these libraries was subsequently cancelled in August of 2003, and the work remains in limbo until another funding source can be procured.

Meanwhile, Repast and Swarm remain incapable of execution on distributed memory parallel computers, though on shared memory parallel computers (which are limited in scalability), there is some capacity for thread-level parallelism. It is becoming increasingly common that ABM model developers complain that they cannot run large enough models on their computers and that they need parallel ABM frameworks, but it remains the case that none are available. The few parallel models in existence do not use frameworks and as such are massive development undertakings that cannot be reused or composed with other simulation components. ABM is still considered a fringe modeling methodology even amongst computational scientists largely because of lack of access to usable scalable tools, and important scientific and national security problems that would lend themselves to ABM are thus analyzed via inferior computational tools.

STRATEGIC ALIGNMENT

EBM represents a core computational capability with potential application to a broad range of multidisciplinary problems. Our proposal seeks to seed a long-term research effort in large-scale EBM within CASC and to position that effort as a central LLNL resource, a role that is difficult for programs to fill. Our focus on HSO and infectious disease modeling reflects lab management prioritization and the most immediately promising avenues for further external funding, but the range and extent of interest amongst other disparate groups at the lab is a strong sign that EBM can be a central modeling resource at the lab. Indeed, at DOE labs Los Alamos, Sandia (Albuquerque and Livermore), and Argonne, similar groups have long had an important presence and role.

REFERENCES

[AdKoChYu99] Adams A., Koopman J., Chick, SE, Yu, P., "GERMS: An Epidemiologic Simulation Tool for Studying Geographic and Social Effects on Infection Transmission," *Proceedings of the 1999 Winter Simulation Conference.*

[AlMc03] Altman, L.K., and McNeil Jr., D. G. "How One Person Can Fuel An Epidemic", *New York Times* article, April 15, 2003.

[CaFrYa02] Carley, K., Fridsma, D, and Yahja, A., "Current State of BioWar and Its Proposed Improvements," *CASOS 2002 Conference*, Pittsburgh, PA.

[FuKu99] Fuentes MA, Kuperman MN, "Cellular automata and epidemiological models with spatial dependence," *Physica A*, 267 (3-4): 471-486, 1999.

[FuLa01] Fuks H. and A. Lawniczak, (2001) "Individual-based lattice model for spatial spread of epidemics," *Discrete Dynamics in Nature and Society*, 6(3):191-200.

[GaLe01] Gani R and Leach S, "Transmission potential of smallpox in contemporary populations," *Nature*, 414 (6865): 748-751, 2001.

[HyLiSt03] Hyman, J.M., Li, J., Stanley, E., "Modeling the impact of random screening and contact tracing in reducing the spread of HIV", *J. MATH BIOSCI* 181 (1): 2003.

[Je87] Jefferson, D, Beckman, B., Wieland, F., Blume, L., DiLoreto, F, Hontalas,P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, Bellenot, "Distributed Simulation and the Time Warp Operating System", *11th Symposium on Operating Systems Principles (SOSP)*, Austin, TX, 1987.

[MiLaAs96] Minar, N., Burkhart, R., Langton, C., Askenazi, M., (1996) "The Swarm Simulation System: A Toolkit for Building Multi-Agent Systems," SFI Paper 96-06-042.

[Wein02] Wein, L., Kaplan, E., Craft, D. "Emergency Response to a Smallpox Attack: The Case for Mass Vaccination", *Proceedings of the N.A.S.*, 99, 10935, 2002 .

[Wo02] Wolfram, S. (2002). *A New Kind of Science*, Wolfram Media, IL, 2002.