Prime Mover: An Event Driven Simulation Framework

Hal Hildebrand

Introduction

Prime Mover is a framework for building discrete event driven simulations. The Prime Mover event driven simulation framework uses Java as the language, runtime and execution framework rather than developing a new language, simulation kernel, or runtime. By reusing, rather than reinventing, Prime Mover leverages the performance, efficiency, ease of use of Java, along with the sophisticated development and debugging environments that Java enjoys. Prime Mover, because it is simply Java, is easy to learn, easy to develop and debug. Prime Mover simulations run anywhere Java does, taking advantage of the extremely sophisticated virtual machine execution environments that Java VMs provide.

The purpose of this document is to provide the user with the information necessary to start using Prime Mover to create their own simulations.

Environment Set Up

This user guide assumes that you are using the Eclipse IDE (Version 3.6.1 or later). The Prime Mover framework has been integrated with Eclipse and running examples within the Eclipse IDE is quite easy.

Prime Mover Feature Installation

The first thing you'll need to do is to install the Prime Mover Eclipse Feature. To do this, add the Prime Mover update site:

http://svn.tensegrity.hellblazer.com/3Space/trunk/update-site/

Prime Mover Project Nature

Once you have added the update site, you should install the Prime Mover feature. When this process finishes, a new menu item will be added which allows you to add the Prime Mover Nature to a Java project - aptly labeled: **Add/Remove Prime Mover Nature**. This is a toggle, so you can remove the Prime Mover project nature.

When an Eclipse project has the Prime Mover nature, the compiled Java classes within the project are transformed using the Prime Mover transform. This process will be explained in more detail later, but for now the important point to understand is that this transformation of the Java classes in the project are automagically transformed by the Prime Mover incremental project builder which is added by the Prime Mover Nature.

Runtime Libraries

The Prime Mover Eclipse feature only includes the Prime Mover transformation logic, and does not contain the Prime Mover runtime kernel. Thus, you must download the Prime Mover runtime jar:

Prime Mover Runtime Kernel

Add this library to your Eclipse project to make use of the Prime Mover framework.

You can also download the sources:

Prime Mover Runtime Kernel Sources

With the Prime Mover Eclipse feature, the Prime Mover runtime library, you will now be set to start exploring event driven simulations using Prime Mover.

Obtaining the Demo

The demo code is available in binary, source and as a Maven project.

Demo Binary:

Prime Mover Demo Binary

Demo Source:

Prime Mover Demo Source

Maven Project

Prime Mover Demo Maven Project

Please download the demo binary or source jar, or check out the SVN Maven repository source so you can follow along with the rest of this user guide.

Discrete Event Driven Simulation

Discrete event simulation is a way of modeling systems that is based on events operating on simulation entities. Event driven simulation models are used to verify designs, create systems for learning or management, or as a simulation infrastructure for virtual reality, augmented reality or computerized games. A discrete event simulation is a system in which the state of the model changes only at a discrete set of simulated points in time. These discrete changes are modeled as simulation *events*.

In an event driven simulation, events are used for communication between the simulated objects as well as used for the modeling an object's behavior. In an event driven simulation, an object will *only* act when an event is produced and "sent" to that object. When an object needs to communicate with another object, the object generates an event and that event is "sent" to the target object. The receiving object could choose to act or change its behavior due to the event *arrival*. Note that only an object that receives an event may generate more events. When an object needs to change its own behavior, the object generates an event that is simply targeted to itself.

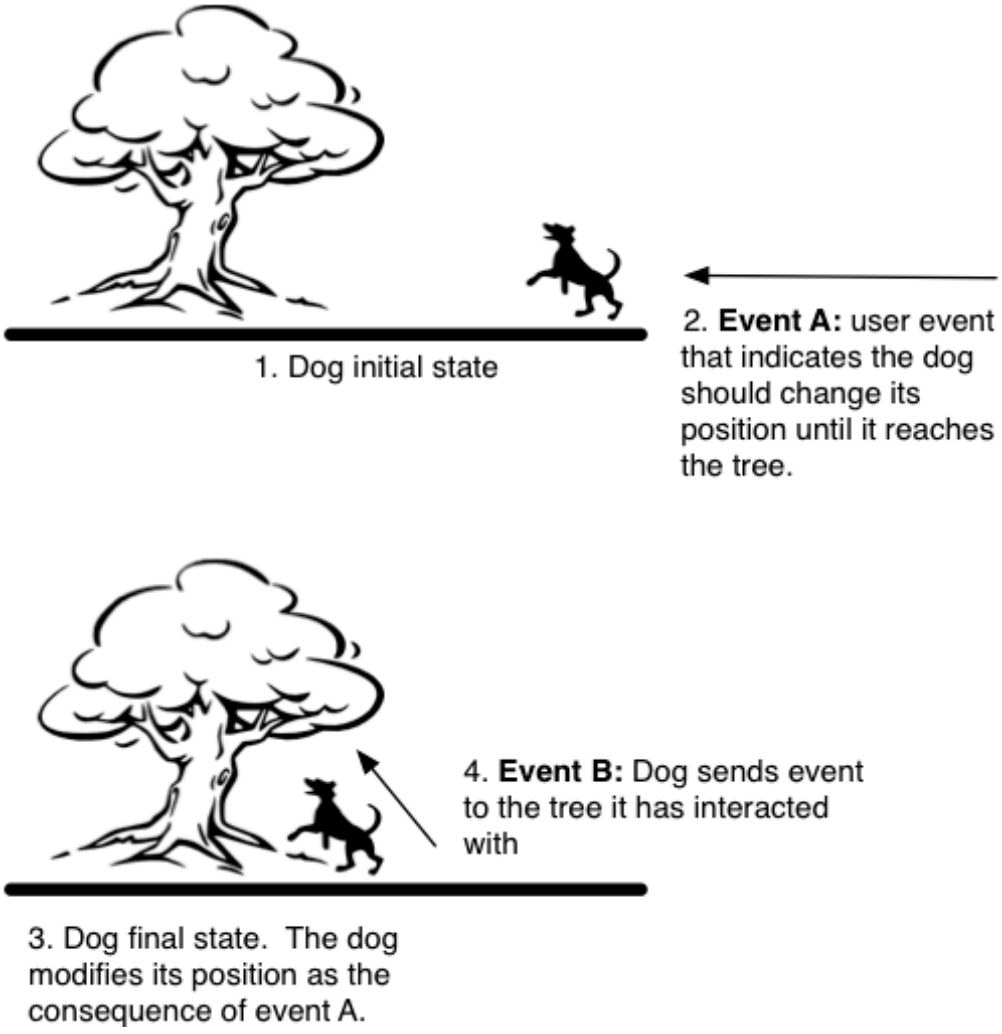
The parallels with object oriented systems should be obvious. In OO programming languages such as Java, entities are modeled with object instances. The behavior of these objects is defined by the class of the instance. These classes define methods which provide the animation of the program. A defining characteristic of OO programming languages is that the interaction between objects is strictly limited to message passing. An instance can send messages to other instances, and an instance can only send messages when it receives a message. An object may also send messages to itself.

Thus, one can think of an event driven simulation as a system of objects which interact with each other via the exchange of events, or messages. A simulation entity's class defines the events that instances can react to in the form of method signatures. The implementation of these events - or methods, depending on your frame - determines the

behavior of the instance and these implementations are themselves nothing but a sequence of message - or event - sends.

Thus, a simulation in Prime Mover is modeled in an object oriented fashion. A simulation is defined by the simulation entities - objects - and the events these objects respond to. Events are generated by the system, from other entities, through scripts, or via external systems such as game controllers or other interaction with external systems.

A Simple Simulation Example



Simulation Events and Entities

In the figure above, there are two simulation entities: the dog and a tree. In the initial state, the dog is located some distance away from the tree. In response to user keyboard input, an event is generated indicating that the dog should change its position to enable it to interact with the tree. The dog receives this event and modifies its position as required by the event. After the dog has reached its final position, the dog generates another event which is directed at the tree, informing the tree that it has been pee'd upon by the dog. In response to this event, the tree may trigger an animation in

which the part of the bark that the dog interacted with becomes a darker color shade, spreading across the side of the tree facing the dog.

One would note that the this model is almost identical to the way objects interact in Java. Objects can only react when they are sent messages, and objects interact with each other only by sending messages to each other. An object may send messages only in response to a message. The key difference in an event driven simulation is that the events (messages) are *scheduled*. In Java, messages (events) are sent immediately. This change in the semantics of how messages that represent events are scheduled and executed is what the Prime Mover event driven simulation framework provides.

Elements of an Event Driven Simulation

At the 20,000 foot level, an event driven simulation consists of three actors:

- The **Simulation Controller**
- The simulation **Entities**
- The simulation **Events**

Simulation Controller

The simulation controller is, not surprisingly, at the heart of the event driven simulation. The controller maintains the simulation clock that defines the current simulation time. Along with the simulation clock, the controller maintains an ordered list of simulation events. It is the controller's responsibility to execute these events at their scheduled execution time.

An event simulation is executed by the event evaluation loop of the controller:

1. Removing the head of the ordered list of enqueued events
2. Setting the current time of the simulation to the event's scheduled time
3. Executing the dequeued event

Entities

As should be obvious now, entities are the nouns in the simulation. They are the things that the simulation model is composed of. These entities have classes which define the events that the entity can respond to.

Events

Likewise, events are the verbs in the simulation. In Prime Mover, these are the methods that an entity defines and inherits.

Simulations

A simulation can thus be viewed as an object oriented model. The execution of the simulation is the execution of the object model. In Prime Mover, execution of a simulation is simply the execution of the Java program that implements the simulation.

So if a Prime Mover simulation is simply a Java program, then what does Prime Mover do and why would one need it?

Hello Sim

Rather than explain what the Prime Mover simulation framework does, let's jump right in and see what a "Hello World" simulation looks like in Prime Mover. The class below is a simple simulation consisting of a single entity and a single event:

```
package hello;
import com.hellblazer.primeMover.Entity;
import com.hellblazer.primeMover.Kronos;
import com.hellblazer.primeMover.SimulationException;
import com.hellblazer.primeMover.controllers.SimulationController;

@Entity
public class HelloWorld {
   public static void main(String[] argv) throws SimulationException {
      SimulationController controller = new SimulationController();
      Kronos.setController(controller);
      new HelloWorld().event1();
      controller.eventLoop();
   }

   public void event1() {
      Kronos.sleep(1);
      event1();
      System.out.println("Hello World @ " + Kronos.currentTime());
   }
}
```

In this example, the simulation entity is represented by the class HelloWorld. The Java annotation, *com.hellblazer.primeMover.Entity* is used to mark the class as a simulation entity. The public, void method *event1()* is the only event on this entity.

The main method of the class first creates a controller for the simulation - in this case, the aptly named *SimulationController* class. This controller is then set as the simulation controller for the current thread using the *Kronos.setController()* method. Once the controller is set, a new instance of simulation entity is created and a single event is sent. Finally, the *eventLoop()* of the controller is called, starting the simulation.

The singular event of the entity is a simple action. The entity first sleeps for 1 simulation time unit, then raises the same event for the entity. After raising the event, the message "Hello World" is printed out with the current simulation time.

If you run this class without first transforming it using the Prime Mover transformation, you'll quickly notice that the result is *StackOverflowException*. The reason is that the method *event1()* is recursive - that is, it calls itself. There is no termination condition, so calling this method will result in an endless series of recursive calls until the stack resources of the thread are exhausted.

When this class is run under the Prime Mover transform, you will see quite a different result:

```
Hello World @ time = 0
Hello World @ time = 1
Hello World @ time = 2
Hello World @ time = 3
...
```

Running the simulation results in the message “Hello World” is printed endlessly until you kill the running of the simulation. No stack overflow.

What’s Happening?

Under the covers, the Prime Mover framework has transformed the class *HelloWorld* into a simulation entity and the method *HelloWorld.event1()* into a simulation event. The difference between a method and a simulation event is that the simulation event is *scheduled*, rather than executed immediately. When a simulation event is sent, the event is scheduled for execution at the current time in the controller.

In the example event, you’ll notice that there is a call to *Kronos.sleep()* before the event is recursively raised. *Kronos* is the API used to control the simulation. In this call, *Kronos.sleep()*, we are telling the runtime to advance the simulation controller’s clock forward by 1 event time unit. After this call, events that are sent will be scheduled at this time - i.e. 1 time unit into “the future”, from the POV of the currently executing event. After the future event is scheduled, the “Hello World” message is then printed out and the event completes its execution.

What this simple example shows is that events are not executed in real time, or immediately. Rather the events are executed in *simulation* time. The execution semantic of simulation events are very different from the Java methods that represent them.

Kronos Versus Kairos

In the Prime Mover event driven simulation framework, it is critical to distinguish between two kinds of time

- Actual time
- Simulation time

Actual time is commonly referred to as “wall clock” time. This is the actual passage of time that we all experience. In the HelloWorld example, time progresses for the program, and when run without the Prime Mover transform, the result is that methods are executed in actual time - i.e. immediately, by the Java VM method dispatch. There is no way to schedule a method execution at a given point in the future. When you send a method to an object in Java, the object receives it immediately.

Simulation time is a quite different beast. It is an imaginary time that only progresses when the simulation is told to advance time. In the example, when the HelloWorld simulation is run after the Prime Mover transform, the events are executed according to the advancement of the simulation’s clock. Events are enqueued by the simulation controller and are only executed when the simulation controller’s clock advances to the point where the event is scheduled for execution.

In our example, the *SimulationController.eventLoop()* simply evaluates events until there are no more scheduled events, or the end of the simulation has occurred. In this simple example, the simulation runs forever as there is always another event to process.

In Prime Mover, we distinguish between time as the simulation perceives time and time as it happens in the real world. In Prime Mover, simulation time is referred to as Kronos time. Real time - where we exist and the programs execute in is referred to as Kairos. The difference between these two times is crucial to understand.

In the simulation, time is an abstract - there is no necessary linkage between the passage of simulation time and wall clock time. Millions of years in simulation time can pass during an hour of wall clock time. Conversely, only a second of simulation time can pass during five hours of clock time. It all depends on what the simulation is doing, the cost of the implementation of the simulation and the desire of the simulation's implementor.

So how does time progress in an event driven simulation? Without going into too much detail - there is plenty of good introductory material on event driven simulation already out there - the way the controller progresses time is by the events that it processes. When the simulation controller processes the events in its event queue, the scheduled time of the processed event becomes the current time. After evaluating that event, the next event in the event queue processed, and that event's scheduled time becomes the current time.

Thus, simulated time progresses in discrete jumps. Time is not a continuous function in a discrete event driven simulation. This allows the simulation to process what is essential to the simulation, rather than churning through an infinity of time between the events - points in time - that matter to the simulation. In this way, event driven simulations can be incredibly efficient - they only process what needs to be processed for the simulation.

Events As Continuations

All simulation events in Prime Mover are essentially continuations. Continuations are deferred execution of code - i.e. execution that is "continued" at some point in the future. When an event is sent, a useful model is that the event state is captured at the point where the event was "raised" (method was sent), and the ultimate execution of the method (event) will be continued when the event is scheduled to be evaluated in the simulation.

Some events are non-blocking, and subsequent events in the simulation code can be immediately processed. Other events are blocking, where the simulation requires the event to be processed before continuing on with the subsequent event logic of the simulation. In Prime Mover, blocking continuations capture the current execution context of the simulation when a blocking event is sent. When this event is subsequently processed, the previous context of the calling event is resumed where it left off.

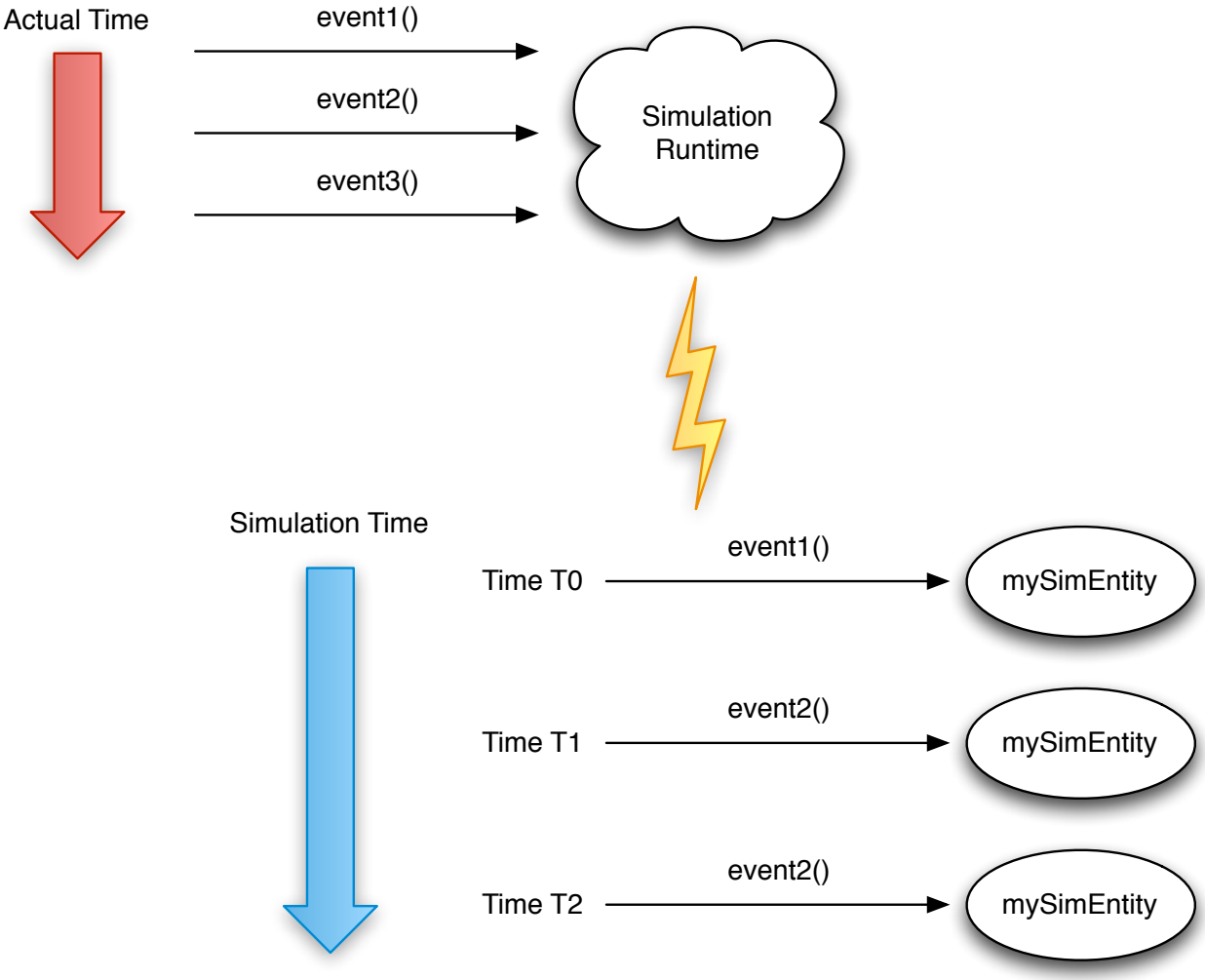
Non-Blocking Events

When creating simulations, one can always use the non-blocking events that Prime Mover provides. Non-blocking events are classical events which do not return a value. In Prime Mover, methods which represent non-blocking events always have a *void* return result and consequently do not return results. When a non-blocking event is sent, the current execution state, in the form of the event's arguments, is captured and queued for execution at the correct simulation time. For example:

```
mySimEntity.event1();
mySimEntity.event2();
mySimEntity.event3();
```

In this example, there are three events raised - event1, event2, and event3. As these are all non-blocking events, these three events will all be scheduled for execution.

In the figure below, the three events are sent, one after another. As these are all non-blocking events, these events are sent, one after another, in actual time. They are, however, not executed in actual time. The Prime Mover runtime defers these events and schedules their execution at times T_0, T_1 and T_2 in the simulation.



Non-Blocking Event Scheduling and Execution

It is important to note that the sending of the three events continue immediately - that is, all three events are raised immediately. The underlying Prime Mover transform and runtime *capture* these events and, rather than executing them immediately, continue their execution at the *scheduled* times in the simulation.

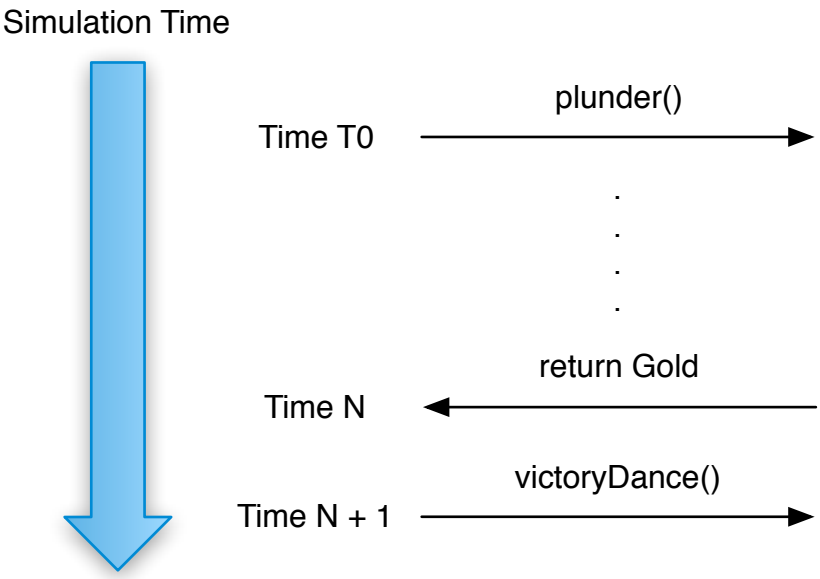
Blocking Events

Non-Blocking events are the bread and butter of a discrete event driven simulation. However, there are circumstances where limiting the expression of a simulation to non-blocking events can result in exceedingly complex code. Relying solely on the metaphor of an asynchronous event can make it difficult to simulate *processes*. Processes are natural models for many simulated systems. When we model processes, blocking operations and transactional exchanges are a natural part of the model. To meet this modeling need, Prime Mover provides *blocking* as well as non-blocking events.

Consider the problem where one entity exchanges information with another.

```
Gold loot = mySimEntity.plunder(0.10);
victoryDance();
```

In this example, 10% of the gold is plundered from the simulation entity. Events which return results are a natural way of expressing transactions between simulation entities. Because a result is returned from the event, we simply cannot defer the execution of the *plunder(float)* event and raise the *victoryDance()* event. We must block the execution of the event until the *plunder(float)* event has been executed so we can return the result and then raise the *victoryDance()* event.



Blocking Event Scheduling and Execution

When using non-blocking events, the distinction between normal Java execution semantics and the simulation event processing semantics is easy to see. For blocking events, the processing of the event is interrupted and is then resumed when the

blocking event resumes. Blocking events are thus true *continuations* in that the currently executing state is suspended and saved for later execution when the simulation schedules the completion of the blocking event.

In our example above, the *plunder()* event it scheduled at time T0 in the simulation. The event which raised the *plunder()* event is blocked until this event executes and then returns - in this example, at time TN. The raising of the *victoryDance()* event is at simulation time T N+1.

If the *plunder()* event was non-blocking, then the *victoryDance()* event would have been raised immediately. Instead, execution is suspended until the *Gold* is returned from the *plunder()* event, and execution continues.

Blocking Event Example

Let's take a look at a simple example which demonstrates the effect and use of blocking events. In this example, we start with the simulation entity *ThreadedExample*. This class has a single event, *process(int)*. In the example, this method is called three times, with the values, {1, 2, 3}. Here's the code for this simple example:

```
@Entity()
public class ThreadedExample {
        public static void main(String[] argv) {
                SimulationController controller = new SimulationController();
                Kronos.setController(controller);

                ThreadedExample threaded = new ThreadedExample();
                threaded.process(1);
                threaded.process(2);
                threaded.process(3);

                controller.eventLoop();
        }

        public void process(int id) {
                for (int i = 1; i <= 5; i++) {
                        System.out.println(Kronos.currentTime() +
                                        ": thread=" + id + ", i="+ i);
                        Kronos.blockingSleep(1);
                }
        }
}
```

When the *process(int)* event is executed, the method iterates 5 times and evaluates a simple loop. Each loop, 3 values are printed out. The current simulation time, the passed in id of the event, and the iteration count.

When properly transformed, running this class results in the following output:

```
0: thread=1, i=1
0: thread=3, i=1
0: thread=2, i=1
1: thread=3, i=2
1: thread=2, i=2
1: thread=1, i=2
2: thread=2, i=3
2: thread=1, i=3
2: thread=3, i=3
3: thread=1, i=4
3: thread=3, i=4
3: thread=2, i=4
4: thread=3, i=5
4: thread=2, i=5
4: thread=1, i=5
```

What this shows is that, in the simulation, the 3 different invocations of the *process(int)* event are interleaved with each other. Because the entity's event calls the *blockingSleep(int)* API, the execution of the event is suspended until one tick of the simulation clock has advanced. In essence, this replicates cooperative multitasking in simulation time, as you can see the three different simulated "threads" interleave their executions in simulation time.

Blocking versus Non-Blocking Events

Of course, it is possible to emulate the implementation of a blocking event with non-blocking events. One can, for example, create a state machine which uses the events to transition the states of the entity as it processes the transaction. The price that one pays, however, is that one has to create a non trivial state diagram which models the non-blocking events that must be exchanged to simulate the blocking event.

Writing simulations using these models is not simply tedious, but it is extremely error prone. The number of states explodes as the number of transactional interactions multiplies in the simulation. Further, the state machines used to implement these transactions are brittle and quickly become a maintenance nightmare, impossible to maintain as the simulation evolves and changes over time. Thus, this modeling technique - while powerful - is not usually suited to those who are not experts at building state machines. Something simpler is needed for the average simulation developer.

Thus, Prime Mover provides the ability raise blocking events. Blocking events are simply events that suspend the execution at the point where they are raised, saving the current execution state of the simulation at that point, and then continue the execution after the blocking event has been executed in the simulation. The notion of a stack that represents the simulation execution state makes using blocking events in Prime Mover a natural way to express transactional events. A stack of suspended execution states is a natural way of modeling the context of a blocking event.

Although all events which return results must by necessity be blocking events, an event which returns no results - i.e. a void return value - can also be a blocking event. Prime Mover allows any event to be marked blocking, whether it returns a result or not.

Concurrency In Simulations

As the previous example of cooperative multitasking “threads” in the simulation has shown, concurrency is easy to accomplish in Prime Mover event driven simulations. And like concurrency in non-simulation systems, it’s important to provide mechanisms to synchronize and coordinate behavior between entities. Prime Mover provides a single primitive, based on the *SynchronousQueue* abstraction in the Communicating Sequential Processes model of CAR Hoare.

The *SynchronousQueue* abstraction is very similar to the *java.util.concurrent.BlockingQueue* The synchronous queue has a capacity of zero, and each insert event must wait for a corresponding remove event by another entity, and vice versa. *SynchronousQueues* are ideal for handoff designs, in which an entity must sync up with another entity in order to hand it some information, event, or task. The *SynchronousQueue* thus provides a simulation time synchronization primitive that can be used to create more complex synchronization primitives, such as locks, semaphores, barriers, monitors, and FIFOs. The *SynchronousQueue* supports both CSP semantics as well as non-blocking sends and receives.

Real Time Simulations

Prime Mover is not limited to implementing simulations where the simulated time is completely disconnected from the wall clock time. An important design goal of Prime Mover is to provide an efficient platform for implementing *real time simulations*, and *scaled real time simulations*.

In a real time simulation, the simulation clock tracks wall clock time. That is, the advancement of the simulation clock matches the progression of real time. Events will only be processed when the wall clock time has progressed to match the scheduled simulation time.

Real time event driven simulations are especially useful in the area of virtual reality and computer games. Event driven simulations have significant advantages over traditional continuous simulation techniques found in virtual reality, augmented reality and gaming simulations. These complicated simulation systems combine a number of extremely complex subsystems that all interact in likewise complex fashion to produce the desired behavior.

In these simulations, sophisticated kinematics models interact with physical animation models which are driven by complex artificial intelligence subsystems. The entire result is continuously rendered in a three dimensional graphics scene. Coupling these complex subsystems is extremely challenging. Getting these sophisticated simulations to take advantage of the multi-core processors in today’s computer systems can be exceedingly challenging. Scaling these complex simulations across many distributed computers is likewise a considerable challenge.

By basing these simulations on a discrete event driven model, rather than the traditional continuous simulation model, these complex subsystems are easier to integrate. Event driven simulation infrastructure provides a natural event driven architecture. This event driven architecture provides a much cleaner and simpler model in which to integrate

these complex subsystems. The result is a much cleaner framework with far looser coupling between the subsystems.

The design of these simulations requires far less computational power to run. Further, because the components are asynchronously coupled through events rather than blocking procedure calls, it is much easier to parcel out the execution of these simulations across multiple threads and across multiple processes and across multiple host machines.

Problems and Challenges in Continuous Simulations

Efficiency in continuous simulation architectures is a big concern. In a continuous simulation, all the entities in simulation are checked, although many entities will never generate events. Some real time continuous simulations only sample the active objects during the simulation phase. Entities that change from activity to inactivity must always be checked in order to verify if they have correctly changed their state. Access all the entities in the simulation when many entities will never generate events, is quite inefficient.

In a continuous simulation architecture, the sampling frequency is common to all the simulation entities, independent of their requirements. The system sampling frequency must be high enough to simulate properly every object to match the Nyquist-Shannon theorem. The upshot is the sampling rate of the framework is driven by the entities requiring the highest sampling rate. If the entities are uniform, this is not a problem. However, it is quite common that the simulation entities have vastly different sampling rates and can easily change their sampling rate requirements during their lifetime.

An under-sampled object could not be simulated properly (losing events or not detecting collisions). Objects with very slow behavior may be oversampled, uselessly wasting valuable CPU cycles. Trying to optimize the architecture to effectively deal with different entity sampling rates adds complexity to the architecture, requires more space to for the management structures (e.g. separate entity lists, categorized by sampling frequency), and compounds multithreading issues in the maintenance of these systems.

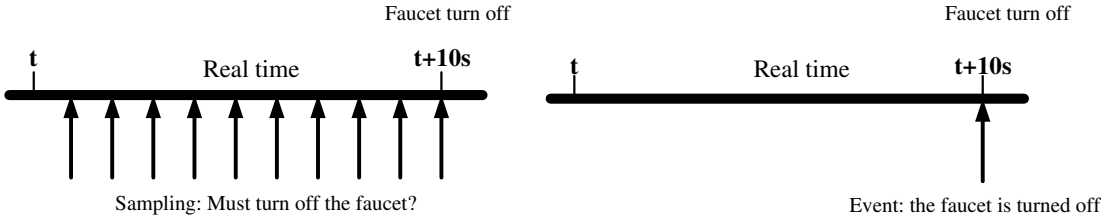
Simulation entities could also have both continuous and discrete behaviors. The continuous system evolution in time may be altered by events not associated to the sampling period. Virtual reality and gaming simulations are therefore invariably hybrid systems. As they are usually implemented as continuous coupled systems, they lose performance and may produce erroneous behaviors.

Advantages of Realtime Event Driven Simulations

When using event driven simulations, the quality of the simulation is improved because the events are executed ordered in time and there is no invalid order of event execution. Each entity in the simulation defines its own sampling frequency, and it is defined according to the behavior of each entity. An event driven simulation is sensitive to events that occur lower than the sampling period, because every entity defines its own sampling period. The level of detail of simulation increases: unnoticed events are now simulated. The sampling period does not change depending on external constraints such as the system load or the scene complexity.

Thus, the simulation efficiency increases. Discrete event driven simulations make better use of the host computer resources. These resources saved can be used to better process other simulation parts such as AI or kinematics. Because only the entities that generate events will be processed, this avoids access to the remaining entities in the simulation. Virtual realities or large scale gaming simulations can thus be executed in machines with lower power. Distributed simulations can be run on machines with different computing power showing the same behavior.

For example, let us suppose that a tank that is filling with water. The water tank faucet must be turned off in 10 seconds. The figure below shows the samples of the object faucet in a continuous and in a discrete simulation. The continuous simulation samples the faucet at each main loop step. The continuous simulation asks the faucet at each sample if it must be turned off at the sample time.



Example of scheduled versus sampled event

In contrast, the discrete simulation plans a discrete event for the given moment the faucet must be turned off. The continuous simulation thus wastes CPU with unnecessary samples of the faucet entity. The discrete simulation uses only the minimum CPU to simulate the entity properly.

Real Time Simulations in Prime Mover

The only mechanism needed to convert your Prime Mover simulations into real time simulations is to use the appropriate simulation controller. Prime Mover supplies an example of a real time controller in the aptly named *com.hellblazer.primeMover.controllers.RealTimeController*. Simply construct a new instance of a RealTimeController, and set this instance as the thread's current controller:

```
RealTimeController controller = new RealTimeController();
Kronos.setController(controller);
```

The controller can be started and stopped with the *start()* and *stop()* methods.

The RealTimeController provided has a resolution of 1 ms, meaning that the controller can't distinguish between events that occur on a sub millisecond granularity. The Prime Mover framework will correct this in future releases by providing a real time controller with nanosecond resolution.

It's important to note that when you use the real time controller, the value of the simulation clock - and thus the timestamps of simulation events - now has a concrete meaning. As the simulation clock can only progress by one tick, and this one tick represents a millisecond, scheduling logic needs to be careful to maintain consistency

between the raw ticks and how these translate to wall clock time. So an entity's event sleeps for one simulation clock tick, the event will sleep for 1 ms.

Note that the RealTimeController is not a magical entity. If you are processing events that, for example, take 20 minutes to execute each processed event, then the simulation is simply not going to progress in real time. Rather, the simulation will continually fall behind real time, losing more time with each bloated event that it processes. This is because the controller executes events in a single thread. This is a design limitation, and there is no way to "add threads" to the simulation to increase parallelism.

Parallel and Distributed Simulation with Prime Mover

This is not to say that this cannot be done. The architecture of Prime Mover was explicitly designed to allow multiple simulation controllers to execute in parallel. While this capability is not exposed, the upshot is that if entities are assigned an "owning" controller, then the entity's events will only be processed by this controller. Even though the entities are on separate threads, animated by separate controllers, the events sent by entities in one thread will be processed by the thread of the receiving entity's controller.

This notion can be taken further and one can imagine using multiple processes, each with their own set of controllers and corresponding threads, managing a population of entities. Events that are sent by an entity in Process A to an entity in process B will therefore execute this event on process B. Note that process A and B can be on separate machines without changing this basic model.

This strategy is, however, a bit naive. The above scenario of parallelizing the simulation using multiple threads in a single process works quite well. The failure semantics of a single process are easy to understand, and ordinary Java message sends are fast and - what is more important - consistently the same. When one moves to a multi process simulation model, things get slightly more complicated. Marshaling and unmarshaling of the events must now occur for events that cross process boundaries. Add to the mix the vagaries of interprocess communication technology and one can easily come up with scenarios in which events are dropped, disordered, etc, between processes. Distributing these processes on different host machines causes event more headaches and compounds the failure modes.

Consequently, although the Prime Mover framework supports these sophisticated scenarios, more infrastructure is required to make these parallel and distributed event simulations work in practice. Making these scenarios a reality is a base design goal of Prime Mover, though, so these problems are being actively worked upon.

Architecture

Prime Mover uses the Java language itself as the language to describe and implement simulations. Prime Mover uses Java classes to represent simulation entities and methods on these entities as the simulation events. A Prime Mover simulation is simply a collection of Java classes, some of which are simulation entities.

Entities

Simulation entities in Prime Mover are properly annotated classes. These classes are transformed by the Prime Mover transform to have simulation event execution semantics, rather than the normal Java method execution semantics.

Events

Events in Prime Mover are represented by methods on an entity class. Scheduling an event is simply the calling of a method on an entity. The difference is that, under the covers, Prime Mover has transformed these method calls into simulation events that execute at the scheduled time in the simulation.

Execution Semantics

To send an event to a simulation entity, one calls the method corresponding to the event on the entity in question. For example

```
MyEntity mySimEntity = new MySimEntity();
mySimEntity.event1();
```

In this simple example, the simulation entity is created just like any other Java object and assigned to the variable *mySimEntity*. In the simulation, the entity will receive the event represented by the method *event1()* at the appropriate point in simulation time when the simulation is executed.

Under normal Java execution semantics, the method *event1()* would be sent immediately - that is, the receiver executes the method *event1()* immediately. In Prime Mover the execution of the simulation the method *event1()* is not executed immediately, but is deferred and queued for execution later by the Prime Mover runtime. At the appropriate point in simulation time, the object *mySimEntity* will then receive the method *event1()* and this method will then be executed. The semantic difference between a Java method and a simulation event is simply that a simulation event is a scheduled action who's execution is deferred.

Event Coupling

By representing events as methods, Prime Mover drastically simplifies the complexity involved in the creation of a simulation. A Prime Mover simulation is simply a program that is written in Java and events are simply the sending of messages.

This tight coupling of events with messages implies is that all events in Prime Mover are strongly typed. The source and target of the event is statically typed by the Java compiler and errors are found at compile time, rather than runtime. Representing events as methods means that the events are typed and thus Prime Mover does not require an additional typing mechanism for the simulation events - Java is the simulation language.

For example, a popular method of creating event driven simulation is to create a framework which represents the events themselves. In these models, the simulation programmer is responsible for creating the event instance, property provisioning the event with the correct parameter state, and then enqueuing the event in the proper queue for the event's target object.

As Prime Mover uses the method execution semantics of Java, there is no need to reify the actual event itself - i.e. the method call is the simulation's representation of sending the event to the target entity. Using Java method calls as the event represent allows the coding of the simulation to be incredibly compact. There is no need to create an event, marshall the correct parameters of the event, and then place the event on the correct event queue.

Events are merely message calls, thus there is also no need of a special mechanism which allows simulation entities to "handle" events - i.e. the "handling" of an event is simply the execution of the method that represents the event. The result is that there is very little disconnect between the simulation and the language representation of the simulation. No events to create, cast and manage. No event queues to understand, organize and manage.

Java Execution Environment Reuse

Representing events as methods also means that Prime Mover can reuse the rich debugging frameworks provided by Java IDEs. Simulations can be single stepped, breakpoints can be set, and execution state examined. Simulations can make full use of any Java library and can intermingle normal Java method execution semantics with the simulation event execution semantics. Simulations are easily unit tested and because they are type safe, standard refactoring and other development methodologies directly translate to Prime Mover simulations.

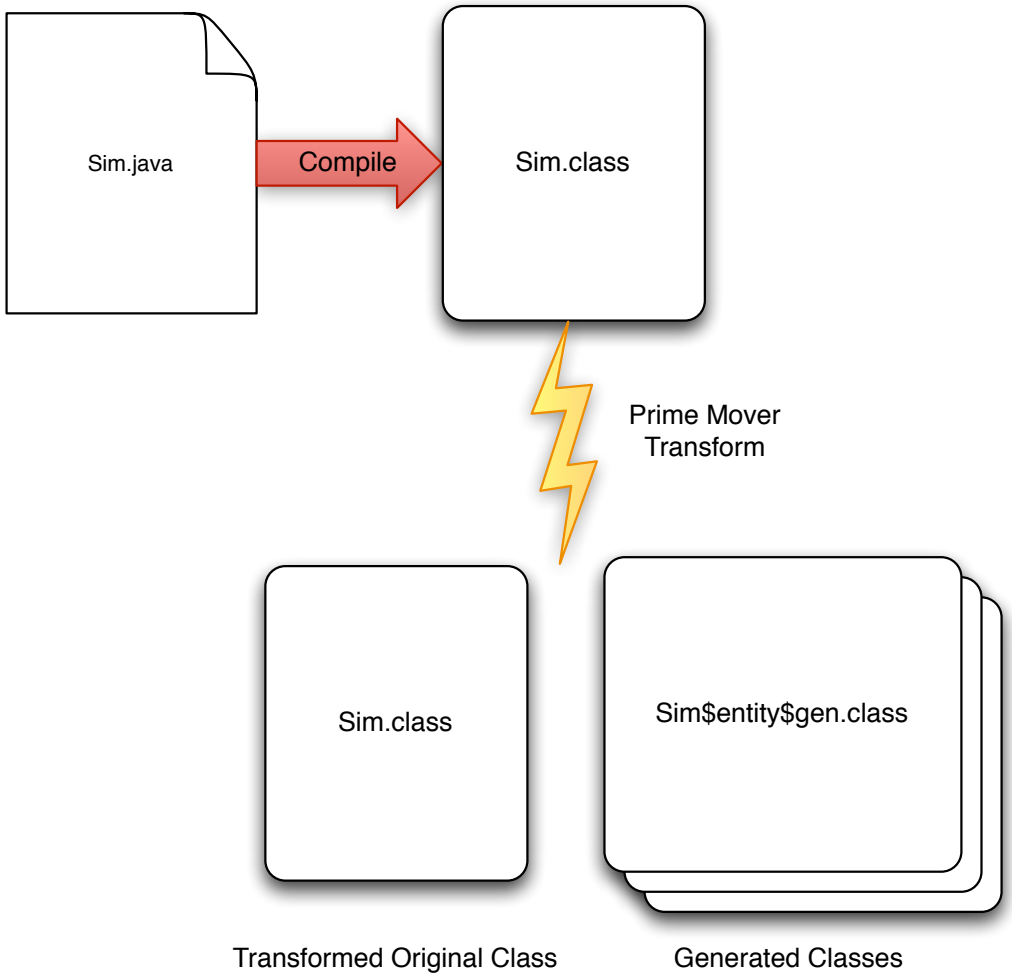
Because Prime Mover uses the underlying Java VM to execute the simulation, Prime Mover inherits all the execution benefits of running as Java code. Prime Mover automatically takes advantage of virtual machine optimizations, the ability to remotely execute code, built in serialization of simulation state, the seamless integration with scripting languages and frameworks, etc.

Prime Mover simulations can also take advantage of existing Java libraries in their implementation. Because regular Java message execution semantics can be freely intermixed with Prime Mover event simulation semantics, the expression of the implementation of the simulation is simple, compact and powerful. Prime Mover allows you to leverage all the accrued advantages of existing systems and implementations when creating your simulations. All done seamlessly and without imposing a heavy cognitive burden on you, the simulation developer.

The Prime Mover Transform

The Prime Mover event driven simulation framework is a byte code rewriting system. This means that the user classes which comprise the implementation of the simulation are compiled by a Java compiler and then transformed by Prime Mover transform to create the runtime simulation.

The Transform Process



The Prime Mover Transform Process

A simulation is simply a set of Java source files. These source files are first compiled by the Java compiler to produce Java class files. The Prime Mover transform then operates on these compiled Java class files. After processing, the original class files are transformed and output by the transform. The transform also optionally outputs a number of generated classes which augment the original classes. This collection of transformed and generated class files produce the compiled representation of simulation.

The Prime Mover transform is a *static* transform and is performed outside the runtime itself, rather than what is known as a *dynamic* transform, where the compiled classes are transformed in a class loader in the Java runtime execution environment.

Static versus Dynamic Transformations

Prime Mover is implemented using a static transform for a number of reasons. The prime reason is simply to not violate the principle of least astonishment. A simulation written in Java and executed in Prime Mover does not act like a normal Java program. Indeed, if you "execute" event simulation code without the Prime Mover translation, it is quite easy to find your programs exploding because of infinite recursion. Combined

with the simulation execution semantics, which result in vastly different execution sequences, it simply is not possible to write simulation code which behaves as anything but erratic when executed without transformation. Thus Prime Mover transforms your compiled Java classes outside of the execution environment so that when run, there is never a surprise in how they execute - other than the inevitable surprises you create for yourself.

Another good reason for using static transformations is that Prime Mover provides some sophisticated code transformations which can take some time to accomplish. In addition to the transformation time overhead, the transformation can also use a large amount of space keeping track of the various pieces of metadata needed to perform the transform. Working with whole sets of programs (or modules, as the case may be), the transform can efficiently reuse metadata and perform a whole system analysis rather than a piecemeal, incremental transform operating on a class by class basis. At the end of the transform, all the metadata used to perform the transform can simply be discarded.

This not only frees up vast amounts of runtime memory which the running simulation can now use, this process is far more efficient. The transform is only done once, regardless of how many processes which run the simulation. Performing dynamic transforms, even if the resulting metadata is discarded, also generates a stunning amount of garbage that taxes a runtime system. By electing to use compile time, static transforms the running simulations quickly start up with no overhead in space and time.

Finally, there is the thorny issues of Java class loaders. Class loaders are extremely powerful entities in Java and current Java VM technology allows for some amazing capabilities. Unfortunately, it's rather tedious and invasive, not to mention error prone, for an embedded framework to take over the class loading infrastructure of an application. Consequently, with a static transform, distributed class files are always correct and will run regardless of whether someone remembers to construct the class loading mechanisms correctly. Prime Mover simulations can therefore be trivially embedded in any Java process, in any application framework such as JEE or OSGi - even frameworks that do their own dynamic byte code transformations at runtime.

Prime Mover API

The Prime Mover API is provided in two packages:

- com.hellblazer.primeMover
- com.hellblazer.primeMover.controllers

The package *com.hellblazer.primeMover.runtime* contains the underlying mechanics that support the running of simulations. This package is internal to the Prime Mover runtime and is not to be directly used by simulation developers.

The two other packages, *com.hellblazer.primeMover* and *com.hellblazer.primeMover.controllers* contain the programatic API exposed to simulation writers.

Package com.hellblazer.primeMover

This package contains the Java annotation classes that mark Java code for the simulation transformation. Two interfaces provide the API to the simulation controller and a simulation entity which can be used to synchronize behavior in simulation time. Rounding out the package is an exception used to indicate an execution error occured during the simulation, and the static collection of APIs to the Prime Mover runtime itself.

Runtime API

The interface to the Prime Mover runtime is represented by the collection of static methods on the *Kronos* class. During the Prime Mover transformation, calls on these methods are rewritten to call the runtime implementation methods on the *Kairos* class. Calling the Prime Mover API methods on *Kronos* without the transform will result an exception.

Time APIs

```
void blockingSleep(long)
void sleep(long)
long currentTime()
```

Kronos defines two methods which simulations can use to advance the simulation clock: *sleep(long)* and *blockingSleep(long)*. The first method, *sleep(long)*, simply advances the simulation clock by the supplied duration and returns immediately. An example of this was seen in the *HelloWorld* example above. The *sleep(long)* method is used to schedule events at a time period in the future - from the simulation's point of view. In contrast, the second method, *blockingSleep(long)*, is treated as a blocking method and the execution will be blocked until the simulation clock has advanced the indicated duration. Finally, Kronos provides the method to return the current time of the simulation with the method *currentTime()*.

Ending The Simulation

```
void endSimulation()
void endSimulationAt(long)
```

These two methods will schedule an event which will cause the simulation to end. The first method *endSimulation()* will schedule the end at the current simulation time. The second is a convenience method which first calls *sleep()*, which advances the simulation clock by the indicated duration, and then calls *endSimulation()*, which then schedules the simulation end at this time.

Creating Synchronization Channels

```
T SynchronizedQueue<T> createChannel(Class<T>)
```

Prime Mover provides a synchronization primitive which can be used to synchronize entities in simulation time. For obvious reasons, the normally used Java mechanisms to synchronize behavior will not work in simulation time. Never the less, it is often necessary to synchronize multiple entities. To this end, Prime Mover provides the CSP *SynchronousQueue*, described previously. The *createChannel()* method is the mechanism used to construct channels during the simulation.

Functional Events

```
void callStatic(long, Method, Object...)
void callStatic(Method, Object)
void run(Runnable)
void runAt(Runnable, long)
```

Sometimes events are not attached to any particular entity and are naturally expressed by static methods. Unfortunately, Prime Mover does not provide a transparent mechanism for dealing with these static methods. However, Prime Mover provides two methods to execute arbitrary static methods as scheduled events. Prime Mover only allows non-blocking static events - no static event may return a value. Prime Mover also provides two methods for executing any *Runnable* instance at a scheduled time in the future.

Simulation Classes

Rounding out the Prime Mover API are three classes which represent the interface to the simulation controller, the interface used to synchronize entities actions, and the exception class used to wrap execution errors raised during the simulation which are not handled by the simulation code.

Controller

The *Controller* interface represents the public API of the simulation controller. The methods of the controller provide the manipulation and access of the simulation's clock, the API for posting events, and the API for setting debug and tracing features of the Prime Mover simulation framework.

Annotations

Simulations are created by annotating classes to indicate these classes are simulation entities, and by annotating methods to indicate these methods are blocking events.

Entity Annotation

The Entity annotation is used to mark a class as a simulation entity. This annotation optionally accepts a list of classes which must be Java interfaces. A simulation entity class defines the events that the entity understands. If no interfaces are provided in the @Entity annotation of a class, then it is assumed that all public methods of the entity are simulation events. Prime mover does not allow methods defined by *java.lang.Object* to represent simulation events, and any methods an entity overrides from Object will not be transformed into events. Prime Mover does, however, consider the methods the entity inherits from superclasses - excluding *java.lang.Object*. The @Entity annotation is inherited - that is, if a superclass is marked as a simulation entity, then any subclasses inherit this property.

When the @Entity annotation is used with interface classes, the only methods which will be considered as events on the entity are the methods defined by the union of the interfaces denoted in the annotation. As the @Entity annotation is inherited, when there are several classes that are annotated with the @Entity annotation, the definition is accumulated through the inheritance chain.

We have already seen the simple example of using the @Entity annotation without indicating any interfaces in the *HelloWorld* example above. The *HelloWorld* class had only a single public method, so this method is transformed into an event. In the example below, the class is annotated as an entity, using the *MySimBehavior* class. From the simulation POV, only the method *simEvent()* will be transformed into a simulation event. The method *nonSimMethod()* will not be transformed into a simulation event.

```
@Entity(MySimBehavior.class)
public class MyOtherSim {
        public void simEvent() {...}
        public void nonSimMethod {...}
}
```

where:

```
public interface MySimBehavior {
        void simEvent();
}
```

Now consider the inheritance case where there are multiple @Entity annotations in the inheritance hierarchy:

```
MySuperSim... @Entity(SuperSimBehavior.class)
        SubSim1
        SomeOtherClass
                MyUltimateSim @Entity(UltimateSimBehavior.class)
```

In this example, we have four classes arranged in a hierarchy. The root of the hierarchy is marked with the @Entity annotation, and thus subclasses of MySuperSIm will all be simulation entities as well. Because MySuperSim defines an interface in the @Entity annotation, the only methods that will be events will be those defined in the SuperSimBehavior. However, notice that MyUltimateSim also defines the @Entity annotation, also indicating an interface class. In this case, the simulation entity now use the union of the two @Entity annotations' indicated interfaces - i.e. both the SuperSimBehavior and UltimateSimBehavior interfaces. Thus, the events defined by MyUltimateSim is the union of these two interfaces.

Blocking Annotation

The @Blocking annotation is used to mark an event as a blocking event. By default, all events which have a *void* return result are non-blocking. By default, all events which return a result are implicitly marked as @Blocking by the Prime Mover transformation. However, it is quite useful to mark a non-blocking method as blocking. In this case, annotating a non-blocking event as @Blocking will allow the Prime Mover transformation to treat it as a blocking event.

NonEvent Annotation

The @NonEvent annotation is used to override an what would normally be considered an event definition by the Prime Mover transform. This annotation is provided for completeness to allow some tricky edge case scenarios to be easily implemented and is not considered to be good practice to use this annotation. It is far better to declare interfaces in the @Entity annotation and use that mechanism to positively identify the events of the entity, rather than to rely on using the @NonEvent annotation to negatively indicate events.

Continuable Annotation

The @Continuable annotation is used by the Prime Mover transform to determine whether a method execution can be continued. Prime Mover uses a byte code transformation technique to implement continuations in the simulation, and to accomplish this, Prime Mover needs to know which methods might call an event which is blocking, or may call another event which ultimately results in calling a blocking event.

Normally, the Prime Mover transform calculates these continuable methods automatically. During the simulation transform, Prime Mover constructs a call tree and symbolically interprets the Java code to determine what methods call other methods. This call tree is then used to correctly mark the methods that need to be transformed into continuable methods.

Due to the polymorphic nature of Java, constructing an accurate call tree can be extremely difficult or - in some cases - impossible to construct. When this occurs, Prime Mover does not have the information necessary to correctly transform the Java class to its simulation form. The @Continuable annotation my be used by the programmer to indicate to the transform which methods are continuable by design.

While it does not hurt the system for the programmer to manually annotate methods, in most common cases, the transform can determine which methods are continuable automatically, thus freeing the programmer from understanding a complicated call tree and maintaining that manually.

Debugging Simulations

Nothing ever goes as planned and an essential part of the simulation modeling process is the debugging of simulations - just as it is with traditional Java programs. Consequently, it is essential that the simulation modeling and execution environment provide a number of rich options which allow the developer to understand what is going on in the simulation, explore the state of the simulation, and trace event executions.

Java Debugging

Because simulations in Prime Mover are simply Java programs, the Java debugger works seamlessly with Prime Mover simulations. Setting a break point in a method will result in that break point being reached when the simulation executes the line of code. As events are simply Java method sends, and simulation entities simply instances of Java classes, all the state of the simulation is available for inspection and manipulation under the Java debugger.

Be aware that these are simulations, not your run of the mill Java code. While the transformations on the Java classes by Prime Mover are faithful to the original code, there are some surprises you'll find as you single step through the execution of your simulation. Prime Mover interposes some generated code in the execution of your original code, and you may find yourself traversing some generated classes which have no source code available. Keep calm and sweep the leg.

At other times you may find the execution of a single line of code requires multiple steps to execute what should be a single method call in your original source. Due to the way that Prime Mover implements blocking events and continuations, you may also find your methods reentered and control flow within these methods a bit different than what you're used to.

Event Logging

Simulation controllers can be provided with a *java.util.logging.Logger* which can be used to log all events that are processed. When the controller is supplied with the logger, the controller logs all event evaluations on this log at the level of INFO.

Event Debugging

Because events are scheduled, rather than executed immediately, the execution of an event is essentially disconnected from where the event was raised in the source code. This nonlinear nature of event driven simulations is unfortunately unavoidable. To aid the debugging of events, the developer can instruct the simulation controller to record the source file location where the event was raised.

This type of recording is expensive, however, as Prime Mover uses the runtime creation of stacks to retrieve this debug source file information. Thus, enabling event debugging can incur a significant performance overhead in the running simulation.

Event Source Tracking

Finally, Prime Mover allows the developer to track the source of events in the simulation. Simulation controllers can be instructed to track the source of an event - that is, the event that caused an event to be raised. These sources form a chain of events - i.e. event C was raised by event B which was raised by event A. This chain of events is the analogy of a stack trace in a normal Java program. Currently we have no event aware debugger for Prime Mover simulations, but if you have event source tracking enabled, you can send the message *Event.printTrace()* to any event to get this event trace.

IDE Integration

Currently, only the Eclipse IDE is supported by the Prime Mover platform. The transformation is integrated into Eclipse by a feature which provides the trampoline to the generic Prime Mover transformation framework. The Eclipse update site for the plugin is:

http://svn.tensegrity.hellblazer.com/3Space/trunk/update-site/

where you can find the latest version of the feature. The Eclipse plugin is available fore Eclipse versions 3.6.1 and greater.

The Prime Mover feature provides three plugins. The first is the Soot optimization framework which is used by the second plugin, the actual Prime Mover transformation framework. The third plugin provides the incremental builder which performs the incremental transformation of the project, and the Prime Mover Nature, which marks the project as being transformed by the incremental builder.

The plugin requires that the Prime Mover runtime library be available as a dependency of the project the Prime Mover Nature has been added to. If the Prime Mover runtime is not provided as a dependency, the transformation will fail and the project will be marked as having an error.

When the transformation is running, useful logging output is provided in a console window. Currently the plug in does not take advantage of the full capabilities of the Eclipse framework and does not provide helpful error markers to annotate your code and provide guidance as to how to resolve the issues that arise in creating simulations. However, useful information is provided by the output in the console and examining its output is useful for understanding what is happening under the hood of the Prime Mover system.

Build System Integration

Maven Integration

The Prime Mover transformation is integrated as a Maven plugin which transforms the compiled classes generated by previous phases. To integrate Prime Mover plugin into your Maven build, first add the 3rdSpace maven repository to your list of repositories - either in your global settings, or in your project's pom.xml:

```
<repository>
   <id>3rdSpace-repo</id>
   <url>http://svn.tensegrity.hellblazer.com/3Space/trunk/maven-repository</url>
   <name>3rdSpace</name>
</repository>
```

To include the Prime Mover runtime dependency, add the following to your list of dependencies:

```
<dependency>
   <groupId>com.hellblazer.primeMover</groupId>
   <artifactId>runtime</artifactId>
   <version>0.0.4-SNAPSHOT</version>
</dependency>
```

Note that you MUST have the Prime Mover runtime as a reachable dependency in the classes you are going to transform, otherwise the transform will fail.

To add the transform to your project, add the following plugin to your pom's list of plugins:

```
<plugin>
   <groupId>com.hellblazer.primeMover</groupId>
   <artifactId>maven.plugin</artifactId>
```

```
        <version>0.0.4-SNAPSHOT</version>
        <executions>
          <execution>
            <goals>
              <goal>transform</goal>
              <goal>transform-test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

This plugin with then automatically transform the project's compiled classes and test classes with the Prime Mover transform.

Ant Integration

Ant integration is currently not supported but is planned. I'm not a big Ant user but I've written my fair share of Ant tasks so it'll be coming sooner rather than later.