DEGAS: Discrete Event Gnu Advanced Scheduler

Luke Ludwig
University of Minnesota
Minneapolis, Minnesota
00-1-763-572-4766
luke@lukeludwig.com

Paul Pukite
BAeSystems
Fridley, Minnesota
00-1-763-572-6561
puk@umn.edu

ABSTRACT
DEGAS provides discrete-event scheduling capability to a GNAT Ada program without requiring extra calls to a simulation library. We accomplish this by intercepting all calls destined for the *pthread* library and then rerouting them to the dynamically linked DEGAS library; this allows a developer to switch between real-time and discrete-event modes at runtime in a non-intrusive manner. DEGAS narrows the separation between simulation and real time applications, and has significant implications for software that includes elements of concurrency, synchronization, and time. We foresee applications that go beyond simulations, including executable specifications, algorithm development, and system verification.

Categories and Subject Descriptors
I.6.8 [**Simulation and Modeling**]: Types of Simulation – *Discrete Event*

General Terms
Algorithms, Reliability, Verification, Performance, Experimentation.

Keywords
Discrete-Event Simulation, Scheduling, pthread, GNAT, Ada, Concurrency.

1. INTRODUCTION
DEGAS[1] contributes to the open-source software community a novel discrete-event scheduler with applications in simulations, executable specifications, algorithm development, and system verification. Most scientists and engineers limit the use of discrete-event systems to simulation applications in which the user must interact directly with a simulation library. We show in this paper how real application code can obtain the benefits of a discrete-event system *without* making calls to a simulation library. DEGAS blurs the line between programs developed as simulations and programs developed for real execution.

[1] Degas (Discrete Event Gnu Advanced Scheduler)
Day-gah: Artsy pronounciation, based on the French impressionist painter.
Dee-gas: Scientific pronounciation, the technique of removing impurities by applying heat in a vacuum.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'06, November 12–16, 2006, Albuquerque, New Mexico, USA.
Copyright 2006 ACM 1-59593-563-0/06/0011...$5.00.

The DEGAS library presents an alternative choice for the developer in need of a discrete-event simulation, but more importantly provides an option to developers who otherwise would not consider simulation viable. The library provides what amounts to a non-intrusive plug-and-play insertion of a discrete-event simulation for any Ada program meeting a few preconditions (much like a leak-checking memory allocator replacement). We accomplish this by intercepting all calls made to the IEEE standard POSIX Threads (aka *pthread*) library at runtime and redirecting them to the DEGAS library instead. The DEGAS library provides the same API as the *pthread* library, such that when DEGAS is dynamically linked ahead of the *pthread* library, all *pthread* calls are handled by the DEGAS library. This gives DEGAS complete control over the threading model, which we use to implement a discrete-event scheduler.

We designed DEGAS to operate with GNAT Ada programs on Linux and tested for portability against the SunOS/Solaris and IRIX operating systems. We find that GNAT provides an open-source high-quality Ada compiler and runtime system to the ubiquitous FSF-based GCC suite [1]. The GNAT runtime system uses the well-known *pthread* library on UNIX-like platforms to implement its tasking model, which DEGAS overrides at runtime to provide deterministic discrete event scheduling. We have used DEGAS to run unit-level and system-level regression test suites of simulation of ground vehicles totaling over 50,000 source lines of code.

1.1 Discrete-Event Simulation
The majority of the literature on discrete event systems remains in the context of simulation. The programming of discrete-event simulation has a long history, dating back to at least 1962 [12]. In most cases, a discrete-event simulation models a system as it temporally progresses by instantaneously changing the state of the system at distinct points in time. All pending events -- any one of which can modify the state of the system -- gets evaluated for execution at each distinct point in time. To improve performance significantly, after an event occurs, we can force the simulation to jump forward in time to the next event. In this manner, the clock itself becomes simulated, which enables a time-consuming real world process to execute in a simulation more-or-less instantaneously. Discrete-event simulations have become extremely useful for analyzing many types of systems, including but not limited to: manufacturing plants, airports, computer systems, ecological systems, and space missions to name but a few examples[14].

Usually the developer of a simulation chooses between using a simulation programming language, such as Modsim III [6], or using a general purpose programming language with a simulation library, such as CSIM for C/C++ [14] or Adam for Ada [16]. To obtain the benefits of a discrete-event simulation, the developer has to either use a specialty simulation language or interact directly with the API of a simulation library. If the simulation remains the means to an end, we can easily justify this decision. However, many simulations have a corresponding software program that separately executes in the real world, and also it has become increasingly popular to embed simulations within the real software application, for example, in virtual hardware-in-the-loop and embedded trainers [4][14]. In such a case the end goal becomes the software product itself, which can routinely grow complex. The simulation then exists as a means to evaluate the real system, in either an objective or subjective context. We will discuss this further in the applications area Section 4.

A discrete-event simulation must keep track of the current simulated time and provide a mechanism to advance time forward. We find three approaches to managing the simulation of time in the literature: event-scheduling, process interaction, and activity scanning [5][13]. These approaches, often called world-views, have a significant impact on how we design and implement simulation software.

In the event-scheduling world view, the developer schedules actions by adding them to a queue, and for each scheduled action identifies everything that may happen as a consequence of that action both at the same instant of time and in the future [13]. The scheduler strips an event off of the queue, executes it, and advances time. This approach provides locality of time. Unfortunately, this does not come intuitively to most developers.

The process interaction world view provides locality of object, meaning each thread in a model describes the action sequence of a particular object [13]. One can create an object and have it operate within its own thread, which comes naturally to many developers. A program developed in this way consists of autonomous entities that communicate and synchronize directly with other autonomous entities.

The activity scanning approach serves as a hybrid of the event-scheduling and process interaction world views, and we find it less commonly used.

DEGAS, and most other simulation libraries such as CSIM and Adam, operate in the process interaction world view [14][16]. When using conventional simulation libraries, the developer cannot use the native language's threading model directly, but instead must figure out how to use the library's API to create the equivalent to threads [1]. For typical programs, the intrusiveness of this requirement gets further amplified as a developer needs to learn yet another API. With DEGAS, the developer can use the full power of Ada's tasking model directly and non-intrusively -- a significant advantage over conventional simulation libraries.

2. DEGAS Algorithm

Threading libraries use various techniques to share the processor amongst the threads within a process, giving the user the sensation of multiple threads executing simultaneously. If one or more of these threads dominate the processor in such a way that prevents another thread from executing, you get the problem of starvation – something a priority-driven scheduling policy tries to prevent. Importantly, the DEGAS scheduler does not concern itself with the problem of starvation. Instead, the currently executing thread will continue to execute in an uninterrupted fashion as long as it possibly can without violating time constraints, locks, or other concurrency interactions. The executing thread will continue until one of the following occurs: a sleep call (Ada's delay statement), it must wait on a condition (*e.g.* similar to Ada's entry call), or it comes to a lock (*e.g.*. which Ada's protected type can duplicate). Simply put, the scheduler will not interrupt an executing thread. Rather, when during the course of execution a thread comes to one of the aforementioned situations, the thread will yield to the scheduler and the scheduler chooses the next thread to execute.

The algorithm consists of a scheduler tied together with the basic *pthread* building blocks. We designed it such that the DEGAS library provides the same API as the *pthread* library, so that when we dynamically link the DEGAS library ahead of the *pthread* library it can intercept all *pthread* calls (recall that the GNAT Ada runtime system typically uses the standard *pthread* library to implement its tasking model on POSIX-compliant platforms). This gives DEGAS complete control over the threading model. As the calls to the system clock do not route through *pthreads*, we must also override the `gettimeofday()` method coming from the standard *libc*. Under Linux, the GNAT Ada runtime system makes calls to `gettimeofday()` to determine the current time during execution, while IRIX and SunOS use `clock_gettime()` (which has a finer resolution). By intercepting this call, the executing algorithm can advance time forward at will, a necessary component of a discrete-event library.

2.1 Choosing the Next Context

The scheduler keeps track of threads in the `cntxtList` array, the primary data structure used by the algorithm (Figure 1).

```
typedef struct {
   ucontext_t context;     /* Stores the context*/
   void (*func) (void *);           /* Behavior */
   void * arg;            /* Arguments for func */
   int waiter;  /* Waiting for time to advance? */
   struct timespec wait;        /* Time to wait */
   int finished;     /* Has context terminated? */
} Cntxt;
Cntxt cntxtList[maxThreads];
```

Figure 1. The cntxtList array contains information about individual contexts.

The scheduling aspect of the algorithm lies in the `cntxtYield()` method shown in Figure 2 and Figure 3. In general, a yield method chooses the next thread context to run, which happens whenever an executing thread has come to a situation in which it must wait for time to advance, a lock needs releasing, or a wake-up signal has to arrive from another thread. When this occurs the executing thread enters a spinlock in which `cntxtYield()` gets called each time through the loop. A specialized kind of busy waiting, a spinlock occurs when the thread simply waits in a loop (”spins”) repeatedly checking until a lock or condition variable becomes available. We invoke the `cntxtYield()` method from the following *pthread* methods, each of which contains a spinlock: *pthread*_cond_wait, *pthread*_cond_timedwait, and *pthread*_mutex_lock.

We use a round-robin technique to choose which thread to execute next. We deviate from perfect round-robin scheduling when all active contexts are engaged in spinlock, and we need to decide which context to wake up. As mentioned, the possible spinlocks include (1) waiting for time to advance in *pthread*-_cond_timedwait, (2) waiting for a signalled condition in *pthread*-

_cond_wait, or (3) waiting for the release of a lock in *pthread*-_mutex_lock. If all active contexts are executing in a spinlock state, the scheduler will choose the context with the minimum amount of wait time as the context to execute next (Figure 2 - `findMinWaitingCntxt()`) – either a context spinning inside of a condition variable or a lock. When the context found spinning inside *pthread*_cond_wait or *pthread*_mutex_lock gets signaled from *pthread*_cond_signal, we set the signalled context with a wait time of zero, ensuring that it gets context-switched to before time advances. Once the scheduler has chosen the next context, the current context gets swapped out with the pending thread using the `swapcontext()` method provided by the `ucontext.h` interface.

```
int cntxtsAllSpinning() {
   return (numSpinningCntxts == numActiveCntxts);
}

void cntxtYield() {
   int lastCntxt = currentCntxt;
   /*  round-robin scheduling */
   do {
      currentCntxt = (currentCntxt + 1) %
                     (numCntxts + 1);
   } while (cntxtList[currentCntxt].finished);

   if (cntxtsAllSpinning()) {
         currentCntxt = findMinWaitingCntxt();
         cntxtList[currentCntxt].waiter = 0;
      }
   }
   /* swapcontext is from ucontext.h */
   swapcontext(&cntxtList[lastCntxt].context,
               &cntxtList[currentCntxt].context);
}
```

Figure 2. The cntxtYield method forms the heart of the DEGAS algorithm.

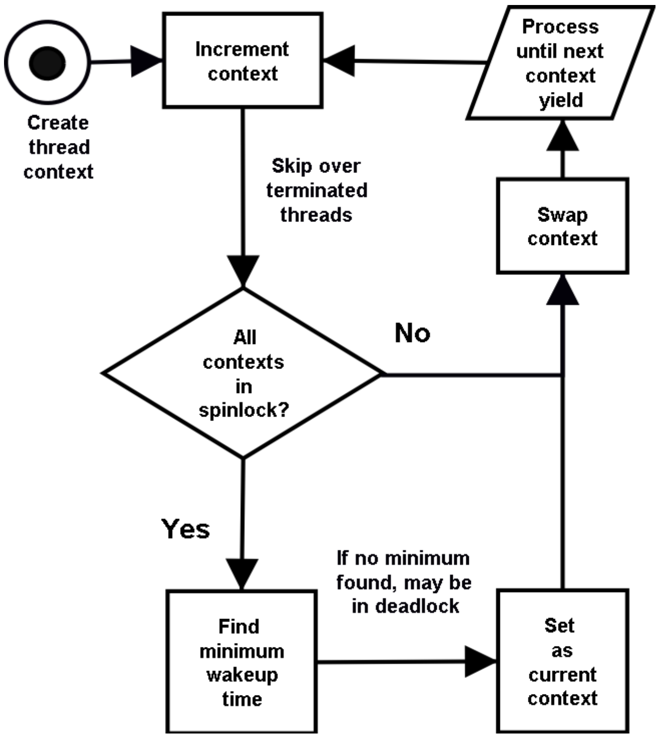
Create thread context
Increment context
Process until next context yield
Skip over terminated threads
Swap context
All contexts in spinlock?
No
Yes
If no minimum found, may be in deadlock
Find minimum wakeup time
Set as current context

Figure 3. Flowchart depicting the algorithm.

We implement the method `cntxtsAllSpinning()` of Figure 2 by tracking the number of spinning contexts and the number of active contexts. The variable `numActiveCntxts` gets incremented each time a context starts and decremented when a context terminates. The variable `numSpinningCntxts` gets incremented before a spinlock and decremented after the spinlock in *pthread*_cond_wait, *pthread*_cond_timedwait, and *pthread*_mutex_lock.

2.2 Implementing the *pthread* methods

Prior to the creation of an Ada task, the runtime system calls the *pthread*_create method. However, instead of executing the normal *pthread*_create method that comes with the *pthread* library, the DEGAS library intercepts the call and replaces it with a context initialization. Inside the surrogate *pthread*_create method, three main activities occur: we allocate memory for a user thread stack, a context gets created using the `ucontext` calls `getcontext()` and `makecontext()`, and we add the initialized `cntxt` object to the `cntxtList` data structure. The `libc` library comes with the standard `ucontext` API, which we use to swap contexts in and out, i.e. user-level thread context switching[2].

As each *pthread* method gets called by the Ada runtime, they correspondingly get intercepted by the DEGAS library. The majority of the *pthread* calls require minimal code. Three of the most crucial *pthread* calls include *pthread*_cond_wait, *pthread*_cond_timedwait, and *pthread*_mutex_lock. These calls cause a task to suspend by either waiting for a condition to occur, a specified amount of time to elapse, or the release of a lock.

The *pthread*_mutex_lock method, shown in Figure 4, gets called by the GNAT runtime anytime it needs a lock, such as to implement some part of Ada's protected type semantics. From the man page on *pthread*_mutex_lock we find three different types of mutexes each with different behavior: fast, error-checking, and recursive. GNAT only uses mutexes of the fast type, therefore, we duplicated this semantic form within DEGAS. According to the man page, if a fast mutex exists in an unlocked state, it becomes locked and owned by the calling thread and the method returns immediately. Otherwise if another thread has locked the mutex, *pthread*_mutex_lock spinlocks until the mutex becomes unlocked. Figure 5 shows our implementation of *pthread*_mutex in terms of a Petri net, a commonly used formal graphical representation of concurrent interactions [10][14].

```
int pthread_mutex_lock (pthread_mutex_t *__mutex)
{
   incrSpinningCntxt();
   while (__mutex->__m_count != 0) {
      cntxtYield();
   }
   decrSpinningCntxt();
   __mutex->__m_count += 1;
   return 0;
}
```

Figure 4. The *pthread*_mutex_lock method (Linux-specific).

[2] At one time GNAT came with a user-level tasking model called FSU threads. As no one maintains this code any longer, it has gone out of favor and remains missing from recent distributions of GNAT.

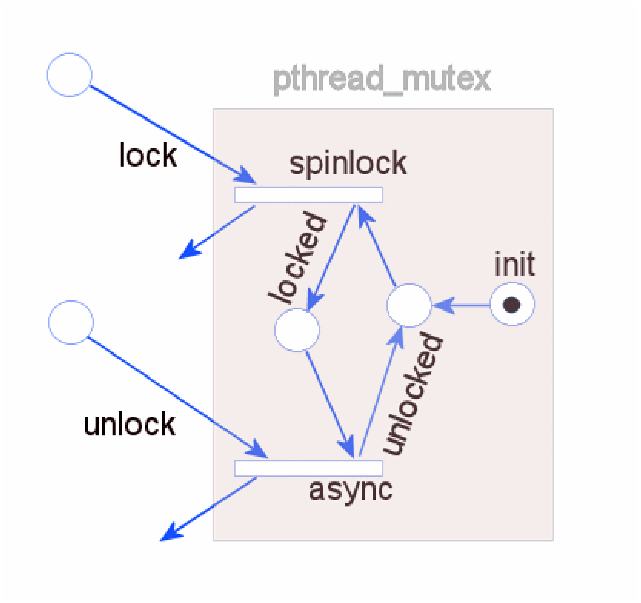

Figure 5. Petri Net for *pthread*_mutex_lock. The call to lock synchronizes on a spinlock scheduling loop. The call to unlock will asynchronously release the spinlock.

The *pthread*_cond_wait method gets called by the GNAT runtime to implement, for example, the tasking rendezvous synchronization pattern. From the man page *pthread*_cond_wait atomically unlocks the mutex and waits for the condition variable `cond` to become signaled. In simple terms, the thread execution loops in a spinlock until the condition variable gets signaled by a separate thread. Each time through the spinlock, the scheduler's `cntxtYield()` method gets called, causing the scheduler to choose a new context to run. Before returning to the calling thread, the *pthread*_cond_wait reacquires the mutex.

```
int pthread_cond_wait (pthread_cond_t *__cond,
                       pthread_mutex_t *__mutex) {
   pthread_mutex_unlock(__mutex);
   __cond->__c_lock.__spinlock = 1;
   incrSpinningCntxt();
   while (__cond->__c_lock.__spinlock != 0) {
      cntxtYield();
   }
   decrSpinningCntxt();
   pthread_mutex_lock(__mutex);
   return 0;
}
```

Figure 6. The *pthread*_cond_wait method.

The *pthread*_cond_timedwait method gets called by the GNAT runtime to implement delay calls. From the man page, *pthread*_cond_timedwait atomically unlocks the mutex and spinlocks until either the cond condition variable gets signaled or the specified amount of time has elapsed. The GNAT Ada run-time only uses the timed aspect of *pthread*_cond_timedwait, so we bypassed the conditional variable semantics. According to the GNAT run-time code, it ignores the return value ETIMEDOUT, assuming it always times out. We advance the global absolute time forward in a monotonic fashion upon leaving the spinlock. Significantly, this remains the only place in the algorithm where time actually advances.

```
int pthread_cond_timedwait (
                    pthread_cond_t *__cond,
                    pthread_mutex_t *__mutex,
                    struct timespec *__abstime) {
   pthread_mutex_unlock(__mutex);
   holdContext(*__abstime, currentCntxt);
   incrSpinningCntxt();
   while (!cntxtList[currentCntxt].waiter) {
      cntxtYield();
   }
   decrSpinningCntxt();
   monotonic_time = *__abstime;
   pthread_mutex_lock(__mutex);
   return 0;
}
```

Figure 7. The *pthread*_cond_timedwait method. Time is advanced forward upon leaving the spinlock by setting the monotonic_time variable.

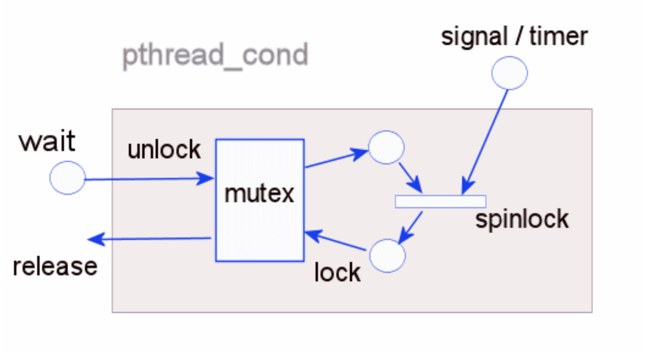

Figure 8. Petri Net for *pthread*_cond_wait and *pthread*_cond_timedwait. For the timedwait call the signal comes from the discrete event scheduler's clock and for the wait call the signal comes from an Ada synchronization primitive.

2.3 Linking DEGAS

On UNIX-like systems such as Linux, applications load the *pthread* run-time as a dynamically loaded library. As a consequence, all call addresses get dynamically resolved through a symbol table lookup. We take advantage of this behavior by loading the DEGAS library ahead of the *pthread* library. This causes the DEGAS symbols to take precedence over the *pthread* symbols. For this to work properly, we need a linker to allow duplicate symbol names.

Since the default GNAT Ada options require that all symbols get resolved at link time, we provide an arbitrary path to the DEGAS library during compilation. Then, during run-time, we can control the loading of the DEGAS library via the `LD_PRELOAD` environment variable (or `LD_LIBRARY_PATH`). If we wish to execute with the DEGAS run-time, we provide a path to the DEGAS shared object library; if we wish to run with the real *pthread* library, we fill `LD_PRELOAD` with a stub implementation of DEGAS which contains no symbols. In the latter case, the real *pthread* symbols get loaded at program startup. In practice, this feature allows us to eliminate additional compile and link cycles from our development process and allows a completely non-intrusive solution.

2.4 The ucontext API

We apply user-level thread primitives to construct the kernel of the discrete-event scheduler. In particular, low-level code stack manipulation calls enable the scheduler to context switch between simulated threads and thus efficiently maintain state. The latter requirement implies that context switching has to obey atomicity, with no repeated or missed code sequences (i.e. no improper jumps). For portability reasons, we did not want to invoke assembly level code to achieve this. Of the possible approaches, the `setjmp/longjmp` and `ucontext` API's showed initial promise. These both supply the non-local goto semantics required to switch between thread-specific code stacks.

In the end, we decided to use the `ucontext` library as it proved very compact and concise. Once we set the context structures with thread-specific data using `getcontext()` and `makecontext()`, the scheduler only needs to invoke the `swapcontext()` call to achieve user-level thread context switching.

We have found the `ucontext` API available on Linux as part of the ubiquitous `libc` library and also available on other Unix platforms, including Solaris and IRIX, as well as an open source implementation for Windows. Unfortunately, variations in the low-level definitions of the *pthread* attributes provides more of an obstacle than the uncontext API in achieving complete portability.

2.5 How Priorities Fit In

A preemptive multi-tasking scheduler considers the effect of priorities in determining which threads to run at any given point in time. Preemption, whereby higher priority tasks get scheduled ahead of a currently executing lower priority thread, serves as a real-world workaround to the difficulty in perfectly scheduling tasks in a non-deterministic environment and for unknown CPU loads. However, a discrete-event-based simulated time scheduler suffers from no such constraint. In fact, every task has an accurate accounting of its time allotment and, in a purely simulated world where time can "stand still", eats no CPU time, so that in the end, we have no need for preemption. Moreover, the only possible need for priorities arises during scheduling "ties" whereby two or more tasks either start or complete at the same time.

In practical terms, we have no need to keep track of priorities when we run software in a discrete-event scheduling mode. This has the benefit of removing degrees of freedom from the scheduler implementation and from the design of simulations, making development much more straightforward and deterministic.

3. TRAFFIC LIGHT EXAMPLE

To demonstrate a trivial example, consider a traffic light simulation consisting of a single intersection with two traffic lights in Figure 9. In this example we have each traffic light represented by a task, which follows the process-interaction world view of discrete-event simulation. We configure the north/south road as the busier road with a green light for 20 seconds, during which the east/west road has a red light. The north/south light changes to yellow for 3 seconds and then becomes red, upon which the north/south task signals the east/west task with the switch rendezvous. Upon receiving the switch signal, the east/west task will delay for 2 seconds, change to a green light for 10, yellow for 3 and then signal back to the north/south task to switch. This continues for the life of the program.

```
procedure Traffic_Light is

   type Color is (Green, Yellow, Red);

   task NorthSouth is
      entry Start;
      entry Switch;
   end NorthSouth;

   task EastWest is
      entry Start;
      entry Switch;
   end EastWest;

   task body NorthSouth is
      My_Light : Color := Green;
   begin
      accept Start;
      loop
         My_Light := Green;
         delay 20.0;
         My_Light := Yellow;
         delay 3.0;
         My_Light := Red;
         EastWest.Switch;
         accept Switch;
         delay 2.0;
      end loop;
   end NorthSouth;

   task body EastWest is
      My_Light : Color := Red;
   begin
      accept Start;
      loop
         accept Switch;
         delay 2.0;
         My_Light := Green;
         delay 10.0;
         My_Light := Yellow;
         delay 3.0;
         My_Light :=  Red;
         NorthSouth.Switch;
      end loop;
   end EastWest;

begin
   NorthSouth.Start;
   EastWest.Start;
end Traffic_Light;
```

Figure 9. A trivial traffic light example, highlighting a key point that we require no calls to a simulation library to run in discrete-event mode. Additionally, a user can switch between discrete-event and real-time modes using the same executable without recompiling.

We can run the traffic light example in either a normal real-time mode or in a hyper-fast discrete-event mode. In the normal mode, time will pass according to the wall clock as expected. One can run the same executable without recompiling in discrete-event mode by dynamically linking the DEGAS library ahead of the *pthread* library. Notice that we require *zero* calls to a simulation library, freeing the developer to use standard Ada, including the standard constructs of rendezvous pattern and the delay call shown in the code snippet. In discrete-event mode, the traffic light program will run as fast as the processor can go. (Obviously, you can only detect this if you place debug statements or include exit criteria in the code to provide data collection end-points.)

4. APPLICATION AREAS

The structure and regularity of the concurrency constructs within the Ada programming language rightly prevents the software developer from using *pthread* synchronization primitives in some arbitrary fashion. An Ada compiler, such as the widely available GNAT, restricts the *pthread* primitives to a minimal set, including create, mutexes, condition variables, and thread-specific storage operations. In turn, each of these gets used in regular patterns guaranteed by the rules of the code generator. Because of this contract between the Ada coder and the intermediate library-level representation, the application designer can depend on an expected set of execution semantics for portable Ada code. This has significant implications for a developer that wants to create software that includes elements of concurrency, synchronization, and time – both in real-time and discrete-event modes

We foresee several prime application areas for an open-source discrete-event scheduler: Simulations, Executable Specifications, Algorithm Development, and System Verification.

4.1 Simulation

The process model for time-based simulation development maps effectively onto a discrete-event scheduler. The simulated clock provides a time base and the Ada tasking constructs lay onto the simulation processes directly. As long as the complete simulation remains monolithic in scope, guaranteeing access to a common clock, the DEGAS run-time pushes the simulation forward in time, with deterministic results. This makes the GNAT compiler mimic, in certain respects, to that of a special purpose simulation engine, such as a VHDL compiler/simulator. We foresee many applications for this mode, with the hyper-speed clock combined with deterministic semantics providing significant benefits. The precision due to the use of a fixed-point definition for Ada's `duration` type also contributes to its effectiveness for many applications, as we can avoid floating-point roundoff errors..

Someone not familiar with the details of the DEGAS approach for simulation may bring up a naïve yet intriguing question: Why not just eliminate DEGAS and speed-up the Ada delay calls by applying a scaling factor? If properly wrapped with a library call and with proper enforcement from coding guidelines, this technique could work in certain situations. For example, a simulated delay of 100 seconds if scaled by 0.001 would take only 0.1 seconds. Unfortunately this approach, if applied uniformly, suffers from several shortcomings. Any delays greater than 0.0 will round to the nearest tic (about 10 msec), thus still taking time, and, worse, the error accumulates for lengthy simulations. Also, since it uses the system's clock, it doesn't give deterministic results. Neither does it work with selective delays and other built-in Ada constructs. In contrast, DEGAS does not suffer from such limitations and one can achieve deterministic resolution to the nanosecond level.

4.2 Executable Specifications

If we intend our simulation to provide a blueprint for elements of a future concurrent software design, we can reclassify our approach as creating an executable specification. For all intents and purposes we don't find a huge distinction between simulations and executable specifications. In general, simulations used as executable specifications typically exercise the logical behavior at the expense of providing insight from randomly generated input data run over multiple trials. In that sense, an executable specification acts as a template from which to substantiate designs before the actual software implementation. In the digital electronics world, VHDL, Verilog, Ptolemy, the "e" verification language [1], and, to a lesser extent, Octave/Matlab serve a similar purpose; in fact, the designers often use these languages to directly generate the net-lists suitable for synthesis into gate arrays and other fabricated logic.

4.3 Algorithm Development

The highest-quality software libraries invariably come about from development processes that involve extensive testing. In general, for algorithms that do not depend on time, you can get the most bang for the buck by simply exercising the software in a unit test environment under exhaustive combinations. Of course this becomes problematic if an algorithm under test references a clock for time. A discrete-event scheduler swaps the real clock for a simulated clock, thus enabling a tremendous potential for speedup. This allows the efficient testing of time based algorithms in a regression test infrastructure. In particular, consider the case of testing for absolute times with dates well in the future. The discrete event solution makes this entirely feasible with little extra effort.

4.4 System Verification

System verification pulls the three preceding application areas together. For example, if we have software application code and enough fidelity to represent the hardware and human elements that the software interfaces to, we can conceivably verify the objective system. For example, we have used DEGAS to run a suite of system regression tests to verify a non-trivial simulation of ground vehicles with a complicated concurrency model.

5. CAPABILITIES

Because the simulated Ada scheduler uses a deterministic clock operating under a closed-world, the execution runs with 100% predictability. In other words, a compiled program, when executed over a set time period, will give exactly the same results when repeated any number of times; even when we include full stack traces and time-stamp logging of individual code statements. On the other hand, a program executed with a real *pthread* run-time will ostensibly give different results, especially in regard to timing of individual code statements. Although a hard-real-time scheduler can reduce the uncertainty, it can't eliminate it. Fortunately, a simulated Ada scheduler totally eliminates this uncertainty.

As a result of the simulated scheduler's determinism, we can apply the algorithm to existing pieces of software and exploit the scheduler's basic features to create novel capabilities. Both the speed and determinism of the scheduler work to our advantage in these cases.

5.1 Stack Checker

As we have to provide our own set of *pthread* stack structures to the simulated scheduler, we can easily monitor potential stack overflow situations. The speed of the simulated scheduler allows us to run through test scenarios much faster than the operating system's scheduler, and since we can potentially capture the stack high-water mark with deterministic precision, we can isolate the problematic parts of the code much more quickly.

5.2 Memory Leaks

By the same token, as we can run application programs at much faster speeds and essentially compress time, employing the simulated scheduler also works to expose potential memory leaks.

For example, a piece of time-based code that may take several days to expose a memory leak, will in practice take much less time, on the order of minutes to find the same leak.

5.3 Model Checking

Because of the non-deterministic nature of most multi-threaded applications, tracking down deadlocks and other synchronization failures remains a daunting prospect to both experienced and novice developers. Like many Ada runtimes, our scheduler has the capability to detect complete deadlocks, but with the advantage of deterministic and repeatable results. Concurrent systems remain difficult to develop and test due to the underlying non-deterministic nature of threading libraries, or as bluntly put by Edward A. Lee, "Threads, as a model of computation, are wildly nondeterministic..." [11]. Lee argues that plain threading remains a poor solution for complex concurrent systems, and suggests that sound concurrent coordination languages [8] are capable of creating concurrent systems that may prove more predictable, deterministic, and reliable.

This portends hope for the future, but unfortunately the world's software repository consists of a fair share of non-deterministic, unpredictable, and unreliable concurrent systems. The state space of possible scheduling paths through a concurrent system routinely turns into an incredibly large number. Standard testing techniques provide limited help since the tests will non-deterministically cover a fraction of this state space.

The most widely used approach to verifying the correctness of a concurrent system involves the use of a modeling language such as SPIN [8]. This requires the developer to create an abstraction of the concurrency model in the modeling language and then this abstraction becomes formally verified by a tool that explores the state space of scheduling paths. This approach consumes considerable developer time and more importantly remains error-prone since the results prove only as good as the abstracted model.

A less commonly encountered approach to verifying the correctness of a concurrent system suggests that we use systematic model checking software such as Verisoft [7]. Verisoft verifies the application code directly, as opposed to an abstracted model of the code. It does this by controlling and observing the execution of the concurrent processes under test by intercepting system calls and driving the code to execute along all possible scheduling paths. In this manner, we can root out deadlocks and other concurrency bugs. In contrast, DEGAS currently routes the application on a single deterministic scheduling path. In the future we plan on extending DEGAS to enable it to check all scheduling paths for systematic verification of concurrent systems in a similar manner to Verisoft.

6. AVAILABILITY

DEGAS consists of around 500 source lines of code (including comments) with less than 200 semicolons. We have made it available on the SourceForge code repository (http://sourceforge.net/) under the *degas* project. On a fast Linux PC it executes at speeds approaching one million context switches per second. We implemented DEGAS in C since both the *pthread* and `ucontext` libraries came with C-based API's and the algorithmic portion of the library code turned out rather concise and short, a case of using the right language for the scale of the job.

7. CONCLUSION

We have shown here how DEGAS provides Ada developers with a discrete-event simulation without having to interface to a simulation library, and provides the ability to switch between real-time and discrete-event modes at runtime. This narrows the separation between simulation and real time applications, and has significant implications for software that includes elements of concurrency, synchronization, and time. We foresee applications not limited to simulations, but also including executable specifications, algorithm development, and system verification. In the near future we plan on modifying the algorithm to work as a general *pthread* replacement, such that DEGAS will provide a discrete-event scheduler for C/C++, python, and other languages that use the *pthread* library. Longer term, we will try to instrument DEGAS to examine all possible scheduling paths in search of concurrency bugs such as deadlock and livelock.

8. REFERENCES

[1] Adya, A., Howell, J., Theimer,M., Bolosky, W.J., and Douceur, J.R., Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming, *Proceedings of the 2002 Usenix Annual Technical Conference.*

[2] AdaCore, Inc. http://adacore.com. 104 Fifth Ave., 15th floor, New York, NY, 10011, USA.

[3] Blasi, A.D., Colucci, F., and Mariani, R. Y-CAN Platform: A Reusable Platform for Design, Verification, and Validation of CAN-based Systems on a Chip. In *Proceedings of the European Test Workshop (ETW 2003)*.

[4] Bounker, P., Brabbs, J., and Adams, C. Low Cost Embedded Simulation System for Ground Vehicles. In *Proceedings of the 1999 Interservice/Industry Training,Simulation and Education Conference* (CD-ROM), National Training Systems Association, Orlando, FL, November 1999.

[5] Fishman, G.S. *Concepts and Methods in Discrete Event Digital Simulation.* John Wiley & Sons Inc, 1973.

[6] Goble, J. Modsim III – A Tutorial. In *Proceedings of the 1997 Winter Simulation Conference (WSC '97),* (Atlanta, GA, USA, Dec. 7-10, 1997). 601-605.

[7] Godefroid, P. Software Model Checking: The Verisoft Approach. *Formal Methods in System Design, 26, 2* (March 2005), 250-255. Kluwer Academic Publishers, Hingham, MA, USA.

[8] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[9] Holzmann, G.J. *The SPIN Model Checker.* Addison-Wesley Professional, Boston, MA, USA. Sept. 2003.

[10] Kavi, K.M., Moshtaghi, A., and Chen, D.J. Modeling Multithreaded Applications Using Petri Nets. *International Journal of Parallel Programming, 30, 5* (Oct. 2002), 353-371. Kluwer Academic Publishers, Norwell, MA, USA.

[11] Lee, E.A. *The Problem with Threads.* Technical Report, UCB/EECS-2006-1, University of California, Berkeley, CA. Jan., 2006.

[12] Nance, R. E. A History of Discrete-Event Simulation Programming Languages. In *Proceeedings of the Second ACM SIGPLAN Conference on History of Programming Languages ('93),* (Cambridge, MA, USA, 1993). 149-175.

[13] Overstreet, M.C and Nance, R.E. Characterizations and Relationships of World Views. In *Proceedings of the 2004 Winter*

Simulation Conference (WSC. '04), (Washington D.C., USA, Dec. 5-8, 2004). IEEE Press. 279-287.

[14] Pukite, J. and Pukite P.R., *Modeling for Reliability Analysis*, Wiley-IEEE Press, 1998.

[15] Schwetman, H. CSIM19: A Powerful Tool for Building System Models. In *Proceedings of the 2001 Winter Simulation Conference (WSC '01)*, (Arlington, VA., Dec. 9-12, 2001). IEEE Computer Society. 250-255.

[16] Sjoland, M, Thyselius, R., and Sjoland, B. Adam, an Ada Simulation Library. In *Proceedings of the Conference on TRI-Ada ('91)*. (San Jose, CA., USA.1991). ACM Press, New York, NY, USA.