Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

EVENT-TRIGGERED ENVIRONMENTS FOR VERIFICATION OF REAL-TIME SYSTEMS

Darren D. Cofer
Murali Rangarajan

Honeywell Laboratories
3660 Technology Dr.
Minneapolis, MN 55418, U.S.A.

ABSTRACT

The growing complexity and the safety-critical requirements of the embedded software in avionics systems present many challenges to current test-based verification technology. The use of formal verification methods can increase design assurance by exploring a larger range of system behaviors and fault conditions than can feasibly be covered by testing or simulation. However, one of the most challenging tasks faced in any formal verification activity is the construction of an adequate model for the environment with which the analyzed system interacts. For real-time systems where the timing characteristics are critical to correct performance this task is even more difficult. In this paper we discuss how an event-triggered model of time (as found in discrete event simulations) can be used as the basis for the environment needed to verify real-time avionics software.

1 INTRODUCTION

Over the past decade Integrated Modular Avionics (IMA) architectures have gained popularity as a more cost-effective method of fielding advanced avionics systems. IMA systems use a shared computing resources to simultaneously host functions of differing criticality, thereby reducing size, weight, power and recurring costs. This makes such platforms a natural location to place new functionality. It also places a special burden on the IMA operating system to keep functions of different criticality levels from interfering with each other.

Currently the primary means for obtaining FAA certification for software-based systems is to develop and test the software in accordance with the guidelines in RTCA document DO-178B (1992). These guidelines emphasize requirements-based testing in the verification process and rely on structural code coverage as a measure of testing adequacy. These structural coverage requirements are not only expensive to achieve, but they can be ineffective in identifying certain classes of errors, especially those involving timing or race conditions.

We believe that techniques and tools being developed in the formal methods community, such as automated model checking, provide practical means for verifying the correct design of complex avionics systems. Model checking is a formal verification technique for finite-state concurrent systems. Unlike testing or simulation, model checking explores all possible behaviors of a system in search of errors.

Our experience with modeling and analyzing the Deos™ scheduler and the ASCB-D communication bus shows that formal methods can be applied to real systems to obtain useful verification results. Using automated model checking can increase design assurance by allowing coverage of a larger range of execution behaviors than can be covered by testing or simulation. Furthermore, model checking can decrease development and testing costs by finding design errors early in the development cycle.

In both the projects described in this paper we have use the model checker SPIN, originally developed at Bell Labs and described in Holzmann (1997). The systems to be verified are modeled using Promela, the input language of SPIN. Promela is a guarded command language supporting asynchronous communication between concurrent computing processes. It is well-suited for describing both software (such as C/C++ code modules) and finite state automata.

In our approach, the system to be verified is modeled with relatively high fidelity and with as direct a correspondence to the actual system as possible. This permits the analysis results obtained for the model to be traced to the actual system. The modeled system must also have some environment or "world" model with which to interact. The environment must generate the system inputs and react to the system outputs. Design of a suitable environment model turns out to be one of the most challenging tasks in formal verification. The environment must be general enough to allow all realistic system behaviors, including any fault conditions that the system is intended to handle. However, the environment should not permit behaviors

that could not actually occur. This includes causal constraints, physical limits, and the passage of time associated with events in the real world. Failure to adequately capture these constraints in the environment model can result in many incorrect counter-examples being discovered by the model checker.

SPIN itself does not model time. The environment model must assume responsibility for the elapsing of time in the model. This manifests itself in the way that the environment determines when to generate particular events and the input values provided to the system model. The timing model we have used corresponds closely to that used in discrete event simulations. At any step in the execution of the model, one or more events are eligible to occur next. The model selects one of these non-deterministically and processes that event. If there is any time associated with the occurrence of the event, the clock is advanced by that amount, and any newly eligible events are added as candidates for the next step.

To illustrate, we next describe two verification problems in which we have used this approach.

2 EXAMPLE: ASCB-D COMMUNICATION BUS

ASCB-D (Avionics Standard Communications Bus, version D) is a data bus designed for real-time, fault-tolerant periodic communications between avionics modules in Honeywell's Primus Epic avionics suite for business, regional, and commuter jet aircraft. The particular algorithm we modeled (Weininger and Cofer 2000) is used to synchronize the clocks of communicating modules to allow periodic transmission of data on the bus.

The synchronization algorithm is sufficiently complex to test the limits of currently available modeling tools. The central performance properties which the algorithm is intended to fulfill are time bounds. It is notoriously difficult to verify that such bounds hold over all possible startup conditions. Furthermore, these bounds must be shown to hold in the presence of numerous hardware faults, some of which are difficult to simulate on actual test hardware.

SPIN proved to be a particularly good tool for modeling such an algorithm; it allowed us not only to verify timing invariants over the state space of the model, but also to conduct random or guided simulations that shed light on the possible behaviors of the model. The graphical representation of these simulations made debugging the model, and eliciting the causes of invariant violations, much easier. Furthermore, the Promela modeling language allowed us to produce an easy-to-understand model that we later found corresponded remarkably well to the C++ implementation code.

2.1 Synchronization Algorithm

The ASCB-D synchronization algorithm is run by each of a number of NICs (Network Interface Controllers) which communicate via a series of buses. These NICs are split into two, corresponding to the pilot and co-pilot sides of an aircraft. For each side there are two buses, a primary and a backup bus. Each NIC can listen to and transmit messages on both of the buses on its own side. It can also listen to, but not transmit on, the primary bus on the other side. From the viewpoint of a given NIC, other NICs on the same side are called *onside* NICs; NICs on the other side are *xside* NICs. Likewise, the buses on the NIC's own side and on the other side are onside and xside buses respectively. The basic structure of buses and NICs is shown in Figure 1.

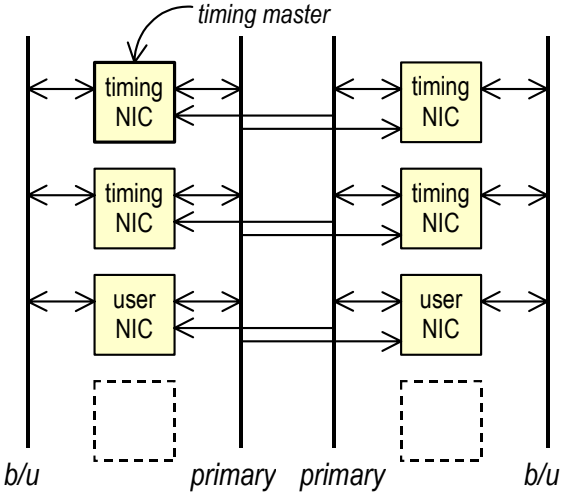

Figure 1: ASCB-D Bus Topology

The operating system running on the NICs produces frame ticks every 12.5 msec which trigger threads to run. In order for periodic communication to operate, all NICs' frame ticks must be synchronized within a certain tolerance. The purpose of the synchronization algorithm is to enable that synchronization to occur and to be maintained within certain performance bounds over a wide range of faulty and non-faulty system conditions.

The synchronization algorithm works by transmitting special timing messages between the NICs. At startup these messages are used to designate the clock of one NIC as a reference to which the other NICs synchronize; after synchronization is achieved, the messages are used to maintain synchronization by correcting for the NICs' clock drift relative to each other. The algorithm is required to achieve synchronization within 200 msec of startup. It must do this regardless of the order in which the NICs start or the time elapsed between their initial startup, and in spite of possible malfunctions in certain NICs or buses.

2.2 Algorithm and Environment Modeling

Our plan for modeling the ASCB-D synchronization algorithm in SPIN was to start with the smallest possible non-trivial subset of the specified algorithm, and increase the complexity of the model in stages from there. We decided to abstract away the numerical time calculations used by the algorithm, and to model only the ordering constraints it imposes. This decision was based on a strong initial aversion to the idea of an explicit model of time. Since the state space of a model must be finite, elapsed time has to be measured in discrete units of some fixed granularity; no matter what granularity is chosen, this strategy is prone to errors caused by lumping two different execution orderings together into the same time unit. Furthermore, since time counters can typically take on a very large number of different values, use of time counters can greatly increase the model state space size.

We found that our initial model with four NICs permitted execution sequences not possible in the real system because the environment process did not have sufficient information to accurately restrict the ordering of the NIC tick events. For example, if the NICs are numbered 0 through 3, the environment would allow ticks and thus message transmissions to occur in a 3-2-1-0-0-1-2-3-3-2-1-0-0-1-2-3 sequence, which is not possible in the real algorithm. Furthermore, the number of possible orderings is so large that is made the model intractable. However, imposing a fixed order on NIC ticks goes to the opposite extreme, excluding many orderings which the real algorithm does allow.

The problems in the four-NIC model necessitated major changes to the basic model structure. The introduction of an explicit numerical time model, and the combination of that time-modeling capability and the message-transmission capability in the same environment process, turned out to be the changes we needed to make the model tractable again.

The time/environment process keeps track of the time remaining until the next frame tick of each NIC and the messages received by each NIC in the current frame, as well as the total time elapsed since start up. It then sits in a loop executing the following algorithm:

```
while(true)
{
   pick id such that timeToNextTick[id] is
minimal;
   send NIC[id] the contents of its message
buffers from the last frame;
   wait for NIC[id] to send back the length
of its next frame, plus the contents of the
message it wants to send;
   if that message is not empty, send it to
the other NICs' buffers;
   for all i != id
     timeToNextTick[i] += timeToNextTick[id];
   total_elapsed_time += timeToNextTick[id];
   set timeToNextTick[id] to the length of
the next frame for NIC[id];
}
```

A number of fundamental changes in the model follow immediately from the introduction of numerical time. These include:

- The environment process neatly encapsulates all those parts of the system that provide input to the algorithm we wish to model (frame ticks, buffers, and buses), while the NIC process encapsulates that algorithm completely. The interface between the two is simple and localized. This is perhaps the most powerful advantage of the time/environment model; it allows faults to be injected and complicated hardware interactions to be added with no change required to the NIC code.
- Because the environment process now dispatches ticks one by one, NICs are trivially guaranteed to execute atomically with respect to each other. NICs also execute atomically with respect to the environment process. This simplifies the space of possible execution orderings dramatically; the only order that matters is the order in which ticks and message transmissions occur.
- Complicated tick orderings produced by frames of different lengths are now explicitly and accurately represented in the model.
- We can now easily test for timing-dependent system properties. For instance, we can place an assertion in the environment process, checked before each tick, that states that all NICs should be in sync within 200 msec of the startup time.
- Because the interface between environment and NIC includes all the data that must be shared between them, there is no need for global data structures. This allows SPIN's compression techniques to reduce the memory requirements.

The reduction in state space produced by eliminating impossible interleavings far exceeds the increase produced by having time counter values.

The ASCB-D modeling effort also taught us quite a bit about modeling systems whose central properties are based on time. Modeling these systems with a purely order-based model introduces large numbers of execution interleavings which do not exist in the real system and which produce spurious violations of safety properties. Furthermore, performance requirements for these algorithms are often expressed in terms of time, so a numerical time model is needed to verify them.

3 EXAMPLE: THE DEOS SCHEDULER

The Deos real-time operating system (Honeywell 1999) was developed by Honeywell for use in Primus Epic com-

puting platforms. Deos hosts many safety-critical aircraft applications, including primary flight controls, autopilot, and displays.

Deos is a microkernel-based real-time operating system that supports flexible Integrated Modular Avionics applications by providing both *space partitioning* at the process level and *time partitioning* at the thread level. Space partitioning ensures that no process can modify the memory of another process without authorization, while time partitioning ensures that a thread's access to its CPU time budget cannot be impaired by the actions of any other thread.

The combination of space and time partitioning makes it possible for applications of different criticalities to run on the same platform at the same time, while ensuring that low-criticality applications do not interfere with the operation of high-criticality applications. This noninterference guarantee reduces system verification and maintenance costs by enabling an application to be changed and re-verified without re-verifying all of the other applications in the system. Deos itself is certified to DO-178B Level A, the highest level of safety-critical certification for avionics software.

The following sections describe key aspects of the scheduler behavior important for our verification work (Cofer and Rangarajan 2002). Then we discuss the challenges faced in developing an appropriate model of time for performing the verification.

3.1 Scheduler Operation

The main components of a Deos-based system are illustrated in Figure 2. A given software application consists of one or more *processes*. Each process is implemented as one or more periodically executing *threads*. All threads in a process share the same virtual address space in memory. Each hardware platform in the system has a separate instance of the Deos kernel running on it. The kernel communicates with its underlying hardware via its hardware abstraction layer (HAL) and platform abstraction layer (PAL) interfaces. The HAL provides access to the CPU and its registers and is considered part of Deos itself. The PAL provides access to other platform hardware, such as I/O devices, timers, and interrupt signals. The application threads interact with the kernel and obtain the services it provides by means of a set of functions called the *application programming interface* (API).

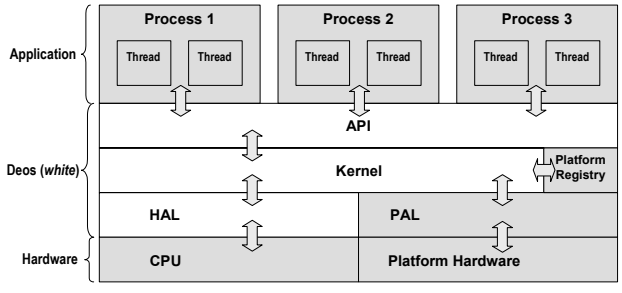

Figure 2: Deos Components and Terminology

Deos must ensure that every application gets its allotted amount of CPU time every period. This provision is called time partitioning, and is accomplished by the Deos scheduler. At system startup, each process is given a fraction of the total available CPU computing resource, called the process's CPU budget, for its use. The process then allocates its CPU budget to its threads. Deos ensures that each of the threads has access to its allocated CPU budget every period.

The Deos scheduler enforces time partitioning using a Rate Monotonic Scheduling (RMS) policy. RMS assigns thread priorities so that shorter period (high rate) threads are assigned a higher priority than long period (low rate) threads. Using this policy, threads run periodically at specified rates and they are given per-period CPU time budgets, which are constrained at thread creation so that the system cannot be overutilized. A thread will be preempted by the scheduler if it attempts to exceed its time budget. Deos threads may also continue run beyond their deadlines if there is unutilized time available in the system using a feature known as *slack scheduling* (Binns 2001).

In Deos, all periods are required to be harmonic. This means that the length of each period is an integer multiple of the length of the next shorter period. Harmonic periods allow Deos to achieve (near) 100% utilization of the available CPU time.

At the start of the thread's period the scheduler places it on a list of threads that are ready to execute in that period. When the thread is at the head of the list for its period, the scheduler starts it by switching the current execution context to that thread. The thread executes until one of three conditions occurs:

- The thread is preempted by the scheduler because a higher priority thread (with a faster rate) is ready to run. The thread is returned to the ready list for its period so that it can be resumed later in its period.
- The thread voluntarily waits for the start of its next period because it has completed its work for the current period.
- The thread uses all of its allotted time and must be stopped by the scheduler to preserve time partitioning. The thread will be run again in its next period and will receive a *threadBudgetExceeded* exception so that it can take any needed recovery actions.

3.2 Scheduler and Environment Modeling

The behavioral properties of the Deos kernel cannot be analyzed independently – the kernel on its own doesn't ac-

tually do anything independent of its environment. The environment in this case consists of:

1. application software threads that invoke kernel services, and
2. platform hardware that provides timers and interrupts to the kernel.

Models in SPIN are specified as collections of concurrently executing processes that communicate via message channels. There are three types of processes defined in our model: the Deos kernel, threads, and the platform environment. The processes and their interconnecting channels are illustrated in Figure 3 and correspond to the components identified in Figure 2.

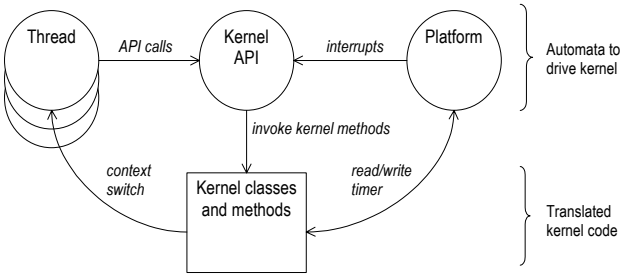

Figure 3: Structure of Kernel and Environment Models

Channels represent various synchronization mechanisms depending on the processes that they connect. Messages sent from threads to the kernel represent calls to services in the kernel API. Messages from the kernel to threads represent context switches to stop and resume thread execution. Messages from the platform to the kernel represent hardware interrupts, either from timers or physical I/O. Messages from the kernel to the platform are used to read and write timer registers. In all cases channels are defined to buffer zero messages, meaning that all process interactions are rendezvous synchronizations requiring the sender and receiver to be ready at the same time.

The Deos kernel process is by far the biggest and most complex process in the model. It consists of a simple state machine that represents a portion of the user API and the translated kernel code itself.

The kernel code that we are modeling consists of C++ classes and methods taken directly from the current Deos software. These structures have been translated into Promela by a very straightforward (though manual) procedure ensuring that the model remains in very close agreement with the actual software.

In the real system, the threads or the platform interface software would directly invoke functions in the kernel API, trapping to the desired service. Since we have modeled these interactions as Promela messages some mechanism is required to invoke the functions in the translated kernel code at the appropriate time. The top level state machine in the kernel model serves this purpose. It first calls all the initialization code in the kernel and then loops in an event handling structure. When it receives messages from the threads or the platform interface corresponding to function calls, it invokes the appropriate kernel function in response.

The kernel can respond to three types of interrupts from the platform environment:

1. System tick (tickintrpt). This is a periodic signal generated by the platform hardware that is used by the kernel to identify the start of all periods. Each period must be a multiple of the system tick rate. On receipt of a system tick, the handleSystemTickInterrupt() method is invoked in the kernel.
2. Timer (*timerintrpt*). This is the time-out signal from the thread timer. It indicates that a thread has reached the end of its budgeted time. On receipt of a timer interrupt the *handleTimerInterrupt()* method is invoked in the kernel.
3. Platform interrupt (*platformintrpt*). This is a user interrupt that may be generated by an I/O device in the system. If the interrupt is not masked, the *raisePlatformInterrupt()* method is invoked in the kernel.

The kernel can invoke the following services in the platform interface, again modeled as Promela messages:

1. Start timer (*start*). This sends a desired starting value computed by the kernel to the timer.
2. Read timer (*getTimeRemaining*). This reads the current value in the thread timer. The timer value is returned as a field in the *TimeRemaining* message sent in reply by the platform process.

The kernel handles the following requests from the threads, each of which is analogous to a function in the kernel API:

1. Create thread (*create*). This indicates that a new thread is to be created by invoking the *createThread()* method in the kernel. The parameters associated with the thread to be created are supplied by reference to a template in the Deos registry. The desired thread template number is specified in the message and passed in as an argument to the API function.
2. Delete thread (*delete*). This indicates that a thread wishes to be deleted from the system by invoking the *deleteThread()* method in the kernel.
3. Wait for next period (*finishedforperiod).* This indicates that the calling thread has completed its work for this period and wishes to be suspended until the start of its next period. On receipt, the *waitUntilNextPeriod()* method is invoked in the kernel.
4. Wait for next interrupt (*waitforintrpt*). This message has been added to support the analysis of interrupt service routine (ISR) threads. Similar to "wait for next period," this indicates that the caller has completed its work and wishes to be suspended until the arrival of its next triggering inter-

rupt event. On receipt, the *waitUntilNext-Interrupt()* method is invoked in the kernel.

The kernel can send the following two messages to the threads to perform a context switch:

1. Stop thread (*stop*). This signals the thread to suspend execution and wait to be resumed by the kernel. This may occur because the scheduler needs to run a higher priority thread or because the thread exceeded its time budget for the current period.
2. Resume thread (*resume*). This signals the thread to continue execution from the point at which it was previously suspended.

In reality, these are not actual requests sent to the threads. Via the HAL, the kernel triggers a context switch causing execution to transfer from one thread to another.

Most of the interesting time partitioning issues are associated with the scheduling interactions between threads. Therefore, our model has a single process consisting of its main thread and several dynamically created user threads. The threads in our model do not have to do any actual work. They provide part of the execution environment needed to analyze the Deos kernel and so they only need to exercise the kernel API in appropriate ways.

3.3 Event-Triggering vs. Time-Triggering

The main choice to be made in the platform environment model is whether the next event to occur will be a system tick or a timer interrupt. This can be accomplished in several ways.

The original version of this part of the environment was developed by NASA Ames (Penix 2000) and evolved with a great deal of experimentation. A set of flags recording the occurrence of various events and clock variables representing the value loaded in the timer, the time used by a thread, the time elapsed since the last tick, and the time remaining on the thread timer are all used to determine what event can occur next and what time should be reported in response to a request for remaining time. This version seems to work quite well but it is very complicated and difficult to modify without introducing errors.

Another approach documented in Pasareanu (2000) uses progressive refinements to develop a correct model of the environment. In this work a universal environment which can generate all possible behaviors without any constraints was used as the starting point. When the kernel was verified in conjunction with this environment counterexamples were produced that were the result of illegal behaviors in the environment. The counterexamples were analyzed and eliminated by adding constraining expressions in linear-time temporal logic (LTL) that capture the illegal behaviors of the environment. This process would be repeated until a genuine counterexample (or completely correct behavior) was achieved.

Based on lessons learned from both of these approaches, we developed a simplified timer environment that accurately captures all of the desired behaviors in the system but is easier to understand and maintain.

An automaton illustrating the time-related activities of the environment is shown in Figure 4. The variable *tick_time* represents the time until the next system tick event while *remaining_time* represents the time until the thread timer expires. If *tick_time* is less than *remaining_time*, the next event that can happen is a system tick, but if *remaining_time* is less than *tick_time* the thread time-out event must occur next. If they are equal either event may be chosen to occur non-deterministically.

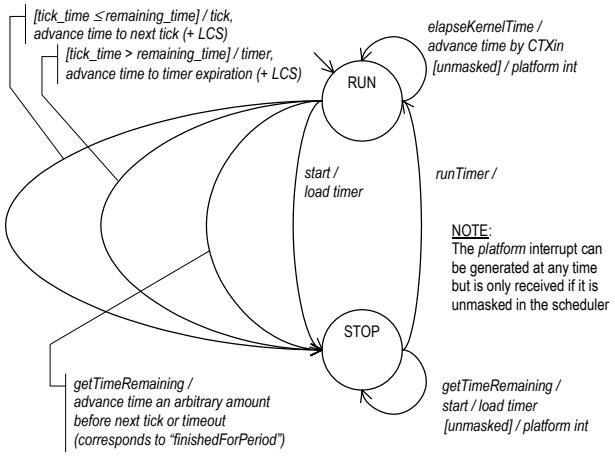

Figure 4: Timer Automaton in Environment Model

On the occurrence of a system tick, time is advanced to the point of the tick event by decrementing *remaining_time* by the value of *tick_time* and reloading *tick_time* with the length of the tick period. On the expiration of the thread timer, time is advanced to the point of the time-out event by decrementing *tick_time* by the value of *remaining_time* and setting *remaining_time* to zero.

When either of these events occurs the kernel will request the value of *remaining_time* by sending a *getTimeRemaining* message. The environment responds with the *timeRemaining* message (which because of the channel synchronization constraint can only be generated if the kernel is waiting for it). In both the tick and time-out case, this value has already been updated thus causing the appropriate amount of time to appear to have elapsed, and no further action is needed.

In the case where a thread has finished for period prior to its timer expiring, some additional calculation is required to cause time to elapse in a reasonable way. Note that in the timer automaton if no tick or timer event has occurred then the *getTimeRemaining* message causes both *remaining_time* and *tick_time* to be decremented by a selected amount. This amount must be chosen such that the finished for period event will appear to occur before the next system tick or time-out event.

The selection of the amount of time that elapses on a finished for period event (the amount of time actually consumed by the thread) is important in determining what behaviors can be realized in the model. In the original timer environment this value was permitted to be either zero, half of the time remaining on the thread timer, or all of the time remaining. This is an approximation of the real behavior possible for a thread but was felt to be a reasonable simplification to keep the state space manageable.

However, there is a problem with this approach, as illustrated in Figure 5. If the time until the next system tick is less than half the time remaining on the thread timer then the only option actually available in the model (such that elapsed is less that the time of the next tick) is to have the thread consume zero time. This does not seem to permit a wide enough range of possibilities for executions in the system. To ensure that there are always three options available in the new timer environment, we compute min(*tick_time, remaining_time*) and allow the selection of zero, half, or all of this value for the elapsed time.

Figure 6 shows a time line for the new timer options. In case 1A *tick_time* is less than *remaining_time* so on a finished for period event the thread can consume zero time,

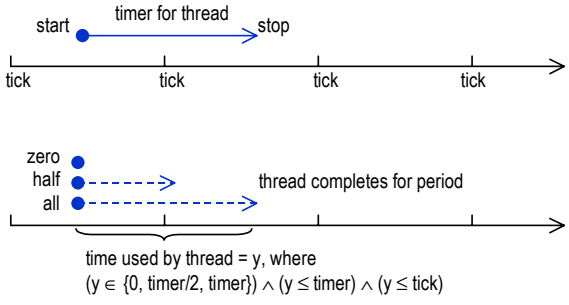

Figure 5: Original Timer Options for Elapsed Time When Thread Completes for Period

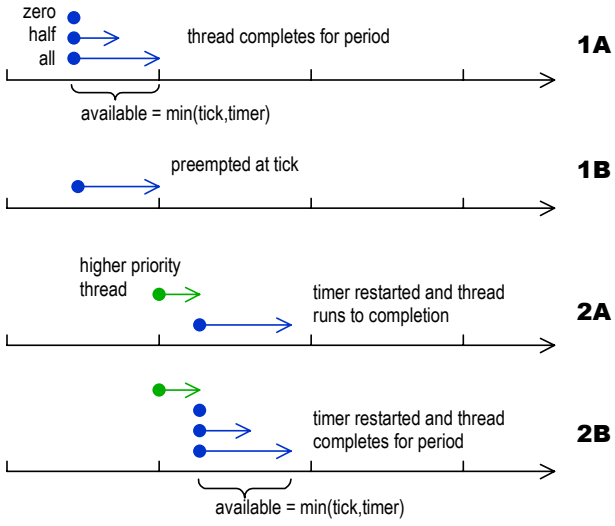

Figure 6: New Timer Options for Elapsed Time

tick_time/2, or *tick_time*. Alternatively, the tick event may occur next and the thread will be preempted by a higher priority thread, as in case 1B. After that thread completes execution, the lower priority thread may be resumed by the kernel. It can either run until its time expires as in case 2A, or finish for period, consuming either zero, *remaining_time*/2, or *remaining_time*, as in case 2B.

3.4 Modeling Scheduler Overhead

Time expended by the kernel performing scheduling or other services on behalf of executing threads is known as *overhead*. Overhead in the kernel arises from a number of sources, some of which are significant as far as their impact on correctness of timing properties while others are not. The terms which we are currently modeling as non-zero are CTXin (context switch in, the time for the CPU to switch to a new thread) and LCS (longest critical section, the time that a tick or timer interrupt might be delayed due to disabled interrupts). The time to handle a system tick is another significant source of kernel overhead. However, its only impact is to reduce to overall system utilization and it is currently set to zero.

The platform automaton includes mechanisms that cause time to elapse at points in its execution corresponding to the actual consumption of overhead in the kernel. Where these times are variable (such as the delay due to a critical section) we have allowed the time consumed to vary (zero/half/all of the maximum).

3.5 Analysis Results

The time partitioning property requires that each thread has access to its CPU budget each period, regardless of the actions of other threads in the system. The simplest way to verify this property is via an assertion embedded in the scheduler model that is checked each time a tick interrupt is processed. The assertion states that for each period completing at the current tick, every thread in the period must either voluntarily complete for period or be terminated due to exhausting its time budget.

Another important set of properties to be verified is associated with preconditions of functions in the kernel. As part of the Deos development process, each function definition must specify any preconditions for the correct use of the function. Where possible, these preconditions have been translated into assertions to be checked by SPIN.

We have augmented our regular verification techniques to include a check for livelock or "non-progress cycles". Checking for non-progress cycles involves placing progress labels in strategic locations in the model. If the model includes a cycle that does not contain a progress label, it signifies the presence of a livelock. In our model of Deos the particular livelock case in which we are interested

is any cycle that does not contain a system tick event, indicating that time is not elapsing as expected.

Our analysis results for the Deos scheduler can be summarized as follows:

- We have been able to verify the main time partitioning assertion and a number of internal function preconditions for many different system configurations, including interrupt threads and slack scheduling. Our overall assurance that this complex system has been designed and implemented correctly is increased as a result.
- We identified several instances where preconditions were inconsistent with the intended operation of the scheduler. These have been corrected and will improve the quality of code reviews performed in future verification and certification activities.
- We identified a number of modeling errors that enabled us to refine the model to reflect realistic system behaviors, resulting in increased confidence in our analysis results.
- We detected several unexpected system behaviors, improving our understanding of the operation of the system. This will be helpful as we maintain and upgrade the system in the future.

4 NEXT STEPS

Our work on modeling and verifying the Deos scheduler is ongoing. We are continuing to add features to the model to increase its fidelity. As the model has increased in size, we have begun work in several new directions to extend our results.

- Model checking suffers from one of the same limitation as simulation in that it requires a specific system configuration must be modeled. We have started work on an more abstract model of the scheduler that permits an arbitrary number of threads and periods to be specified. We are using the PVS automated theorem proving system for this work.
- In our model of time in the environment model, we currently limit thread completion times to zero, half, or all of the available time. We are working on a proof that this time abstraction is indeed correct and does not exclude any real system behaviors.

ACKNOWLEDGMENTS

This work was funded in part by NASA under cooperative agreement NCC-1-399.

REFERENCES

Binns, Pam. 2001. A robust high-performance time partitioning algorithm: the Digital Engine Operating System (Deos) approach. In *Proceedings of the 20th Digital Avionics System*, Daytona Beach, FL.

Cofer, Darren and Murali Rangarajan. 2002. Formal Modeling and Analysis of Advanced Scheduling Features in an Avionics RTOS. In *Proceedings EMSOFT '02: Second International Workshop on Embedded Software*, ed. Alberto Sangiovanni-Vincentelli and Joseph Sifakis. Springer-Verlag.

Holzmann, G. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23 (5), 279–295.

Honeywell. 1999. Design Description Document for the Digital Engine Operating System. Honeywell specification no. PS7022409.

Pasareanu, Corina S. 2000. Deos Kernel: Environment Modeling using LTL Assumptions. NASA Ames Technical Report NASA-ARC-IC-2000-196.

Penix, J., W. Visser, E. Engstrom, A. Larson, and N. Weininger. 2000. Verification of Time Partitioning in the Deos Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland.

RTCA. 1992. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc., Washington, DC.

Weininger, Nicholas, and Darren Cofer. 2000. Modeling the ASCB-D Synchronization Algorithm with SPIN: A Case Study. In *Proceedings of the 7th SPIN Workshop*, LNCS 1885, Springer-Verlag.

AUTHOR BIOGRAPHIES

DARREN D. COFER is a Senior Principal Scientist at Honeywell Laboratories in Minneapolis, MN. He received Ph.D. and M.S. degrees in Electrical and Computer Engineering from the University of Texas at Austin, and a B.S. in Electrical Engineering from Rice University. His research interests include analysis methods and tools for verifying correctness of avionics systems, discrete-event and hybrid control system design, and embedded systems design for safety-critical applications. His e-mail address is <darren.cofer@honeywell.com>.

MURALI RANGARAJAN is a Research Scientist at Honeywell Laboratories. He received his Ph.D. in Computer Science and Engineering from the University of Cincinnati a B.E. in Computer Science and Engineering from the University of Madras, India. His principal area of expertise is in applying formal verification techniques for analysis of complex systems with emphasis on automation. His background includes work with automated application of theorem proving techniques for hierarchical analysis of hardware systems and application of model checking techniques for analysis of software systems. His e-mail address is <murali.rangarajan@honeywell.com>.