Verification of Time Partitioning in the DEOS Scheduler Kernel

John Penix and Willem Visser
Automated Software Engineering Group
NASA Ames Research Center
Moffet Field, CA 94035

Eric Engstrom, Aaron Larson and Nicholas Weininger
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

Abstract

This paper describes an experiment to use the Spin model checking system to support automated verification of time partitioning in the Honeywell DEOS real-time scheduling kernel. The goal of the experiment was to investigate whether model checking could be used to find a subtle implementation error that was originally discovered and fixed during the standard formal review process. To conduct the experiment, a core slice of the DEOS scheduling kernel was first translated without abstraction from C++ into Promela (the input language for Spin). We constructed an abstract "test-driver" environment and carefully introduced several abstractions into the system to support verification. Several experiments were run to attempt to verify that the system implementation adhered to the critical time partitioning requirements. During these experiments, the known error was rediscovered in the time partitioning implementation. We believe this case study provides several insights into how to develop cost-effective methods and tools to support the software design and implementation review process.

1 INTRODUCTION

The Honeywell Dynamic Enforcement Operating System (DEOS) is a real-time operating system for integrated modular avionics systems. Due to the inherent complexity and safety critical nature of the system, the developers understood from the beginning of the DEOS development that testing was going to be inadequate for ensuring the correctness of the scheduler. Currently, the primary means to ensure FAA software certification is to develop and test the software in accordance with the guidelines in RTCA document DO-178B [19] which uses structural coverage as a measure of testing adequacy. However, the structural coverage requirements specified in DO-178B are expensive and ineffective in identifying certain classes of errors, especially those involving timing or race conditions.

To verify the DEOS scheduler, Honeywell employed a collection of techniques including the specification of semi-formal pre-conditions, post-conditions and invariants on C++ functions, data structures, and abstract system states. The formal software development review included checking manually that the pre-conditions, post-conditions and invariants were satisfied by the implementation. During several of these reviews, very subtle errors were detected that the developers believed would have been impossible to detect without these techniques. The developers are confident that introduction of these techniques greatly increased the safety of the DEOS implementation and they became interested in increasing the formality, reliability and efficiency of the review process by using automation.

This paper describes a collaboration between NASA Ames and Honeywell to investigate techniques that might enable automated tools to be employed as part of the software development process. To determine the efficacy of automated techniques, a "slice" of the scheduler code including one of the most subtle errors detected during the DEOS development was selected by Honeywell and delivered to NASA for analysis. The slice contained 10 classes and over 1000 lines of actual code, without comments. A one-day overview of DEOS was made to the NASA and it was indicated that the given system failed to maintain the time partitioning in the presence of dynamic threads. However, this did not provide insight into where the fault might be in the code.

The specific technique investigated was *model checking*, a formal verification technique for finite-state concurrent systems [4, 6, 14, 18]. Model checking shares some characteristics with testing in the sense that it is highly automated, but unlike testing, it examines all possible behaviors of a system in search of errors. Model checking is specifically designed to find errors in concurrent software that are difficult to find using traditional testing, such as race conditions and deadlocks. This report details the process and results of applying model checking to the DEOS scheduler slice.

There are several hurdles that must be overcome to make model-checking practical for program verification. First, any inputs to the model checker must have a direct and easily maintainable correspondence with standard software development artifacts. Second, an environment must be constructed to drive the program. Finally, the size of the model's state space must be reduced, usually via abstraction, to permit exhaustive verification within practical memory limitations.

Common practice in model checking is to hand translate the original software to a form that can be handled by a model

checker. This translation usually incorporates a significant amount of manual abstraction to reduce the size and complexity of the software. To more closely integrate model checking with the software development process, we propose to have model checkers take program source code as input, removing the time/effort spent on translation [13]. This also increases the level of assurance in the translation, which is a weak link in the verification process. The first step toward automated translation is the development of a systematic translation process. Section 3 describes a systematic source level translation of part of the DEOS operating system kernel into Promela, the input language for the Spin model checker.

To model check (or test) a system, an environment must be constructed that drives the program [10]. In the case of DEOS, it was necessary to construct models of the possible behaviors of user threads, the system clock and the system timer. Section 4 describes the environment that was constructed for verification of the kernel. To reduce the size of the state space, the environment model used for verification contains a significant amount of abstraction with respect to the modeling of time.

A fundamental barrier for model checking is that the size of the state space that needs to be searched during verification grows exponentially with the number of processes in the system. In practice, this means that only limited coverage can be achieved when checking for errors. Section 5 shows that, even in cases where verification cannot be done exhaustively, useful analysis can be done on the system.

One approach to managing state space explosion is abstraction. It is imperative to understand how abstractions effect the validity of the property under investigation, so abstraction should be applied in a controlled framework [5]. In this experiment, we separated translation from abstraction as much as possible, to provide a cleaner framework for experimenting with different abstractions. We perform abstraction on the program (as opposed to the model) and then perform translation. This approach allows information about the program and program abstraction and specialization techniques [11, 12, 20] to be used to support the method. We think this approach is more likely to be usable by programmers who are familiar with the application being verified but unfamiliar with formal specification languages. Section 6 describes our application of predicate abstraction [11] to allow exhaustive verification of the DEOS kernel. We believe that our abstractions of timer and counter variables in DEOS may be applicable in other real-time software.

2 OVERVIEW OF DEOS

DEOS is a portable microkernel-based real-time operating system used in Honeywell's Primus Epic avionics product line. DEOS 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 a single 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 possible level of safety-critical certification.

The DEOS scheduler enforces time partitioning using a Rate Monotonic Analysis (RMA) scheduling policy. Figure 1 shows an example DEOS scheduling timeline. In the example, the system contains a main thread, two user threads and the special idle thread which runs when no other threads are schedulable. The main thread runs in the fastest period, and therefore also at the highest priority, with a budget of 5 out of 20 time units. The user threads run in a period 3 times as long at the main thread, each with a budget of 20/60 time units. In the example, all of the threads are scheduled and appropriately allocated their requested budget within their respective periods. Threads are interrupted when they use all of their budget (timer interrupt) or when a thread of higher priority becomes schedulable (preemption). The idle thread runs at the end of the sequence to take up the slack time in the system that is not requested by any thread.

Many real-time operating systems are at least partially statically scheduled, which makes it relatively easy to analyze the possible execution sequences in the system. DEOS, however, supports fully dynamic creation and deletion of threads and processes at runtime. When threads are created within a process, they receive some budget from the main thread for that process. When they are deleted, the budget is returned to the main thread [2]. DEOS also provides a rich set of thread synchronization and inter-process communication primitives. As a result of this complexity, the number of possible interleavings of program execution in DEOS is enormous, and calculations such as schedulability analyses must often be made at runtime. This makes systematic verification of time partitioning a difficult task.

3 DEOS TO PROMELA TRANSLATION

The DEOS Kernel is written in an object-oriented style using C++. In contrast, Promela is a process based imperative language which uses shared memory and message passing for communication. The fundamental translation problem we had to overcome was modeling objects within the framework of processes. The most intuitive approach is to model classes as processes: each object is an instantiation of the corresponding process and method calls are synchronous

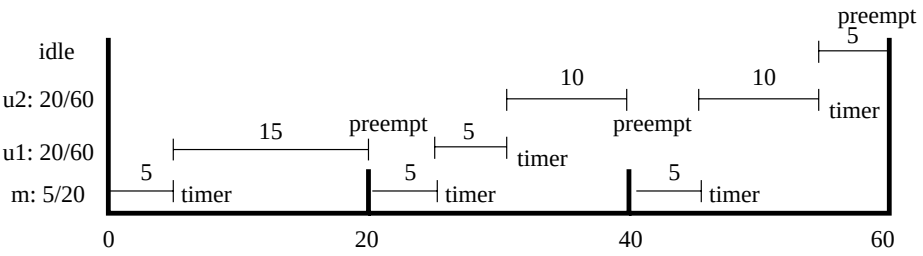

Figure 1: Thread Scheduling in DEOS

messages passed between processes. This is simple and elegant, but is incorrect with respect to the "semantics" of C++ (or JAVA) where more than one thread can call a method of a shared object simultaneously. There is also a state explosion problem with this approach because it introduces an interprocess communication for each method call, which are treated as a points of potential interleaving by Spin.

These problems eventually led us to develop an approach which involved extending Spin [21]. However, we performed the majority of the verification activity using a translation that stored object data in arrays of records. We believe that this translation is of more general interest, so we focus on it here. A more detailed comparison of the different translation approaches is described in an extended technical report [16].

The translation we used is based on modeling classes as records and using arrays of these records to store object data [1]. Using this method, the index to an entry in the array is a reference to an object of the class. An object reference (or pointer) is therefore an integer value, and are declared to be of type **byte**[2]. The translation requires all object variables to be references because it is not possible to get the index (address) of an object. The number of instances of each class must be determined statically to provide array bounds. We also use the array bound as a NULL pointer, which provides an array bounds error in the case of a null pointer dereference. Any `static` data members are declared as global variables.

The methods of a class are modeled by macro expansion with the **inline** construct of Promela, because Promela does not support functions. This provided several challenges. Inline commands do not return values, so it is necessary to pass in the variable that would normally be assigned the return value and do the assignment within the inline. It was also necessary to pass in the index to reference the object data. Promela does not support recursion, because inlines are macro expanded. We worked around this problem by expanding the recursion by hand. This worked in the case of DEOS because the 'recursion' was only used to call the same method in a different object in a non-cyclic pattern.

Using this translation scheme was generally straightforward and did not require understanding the original code. However, we did encounter a problem with inheritance because there was no means to determine the class of an object. For example, suppose class B inherits methods and data from a superclass A. If an object is referenced by a pointer of type A, it may be an A or a B, so we need to keep track of the class of these objects to determine which array contains the object's data. In DEOS, the problem was in the classes that implement the doubly link lists used throughout the system. There is a super class called `DoubleLinkListNode` with `previous` and `next` pointers, and two subclasses: `threadList` that represents a list and `threadListNode` that represents the container nodes that point to the actual objects in the list. Initially it appeared that the two types of nodes were distinguishable. However, we failed to consider the dynamic behavior of the lists, where it was no longer possible to know whether a `next` or `previous` pointer of a node points to a `threadList` or a `threadListNode`.

Our first approach to solving this problem was to extend references to objects to also contain a `Class` field. However, this resulted in code blowup. For example, within `mergeList` the statement

```
previous->next = otherList->next
```

translates to 32 lines of Promela to handle all of the potential combinations of object types. We also found the code produced by this translation to be difficult to read.

In our second approach, we noticed we could flatten the inheritance hierarchy in the case of `threadlist` and `threadListNode`: i.e. we made the two classes one by adding the data pointer to the `threadList` class and removing most references to the `threadListNode` class[3].

[1] A similar technique was used by Havelund and Pressburger for Java [13].

[2] In general, we translate every integer type used within DEOS to the **byte** type of Promela that uses only 8 bits. This is one of the few abstractions that must occur in the translation.

[3] The constructor needed to be retained since threadLists and threadListNodes are initialized differently.

This removed the ambiguity as to where the object's data was stored, and the type correctness of the C++ program assured that a `threadListNode` method could not be called by a `threadList`.

Both alternatives suffer from using excess space. With flattening, there are unused fields in objects (corresponding to member data of other types) that increase the size of the state vector. Using class fields requires additional information in the state vector. However, the class fields approach also requires additional code to determine the appropriate array to access, which increases the state space by adding control state in the program. In the DEOS kernel, the amount of additional code required caused the state space to be increased significantly, leading us to prefer the flattening solution.

4 ENVIRONMENT MODELING

The most difficult task in the experiment was constructing an adequate environment for DEOS to execute in. The kernel must receive calls from the threads that run on the kernel as well as system ticks and timer interrupts from the hardware. The environment turned out to be of crucial importance for achieving meaningful results during the model checking. Specifically, the modeling of time provided challenges in trading off result validity versus state space size.

Figure 2 illustrates the Promela environment we constructed to model check the DEOS kernel. There is a box for each concurrently executing process: the kernel, the idle thread, the main thread, n user threads to be scheduled by DEOS, the system tick generator and the timer process. The dotted box around the last two indicate that we eventually combined the system tick generator and the timer into one process. Communication between the processes is achieved using synchronous message passing, which is illustrated by the labeled arrows in the figure, as well as the Promela code of the kernel. Dotted arrows indicate values being returned to the calling process. In the following sections, we discuss the different components of the DEOS kernel and its environment in detail.

The DEOS Kernel

To allow the kernel code to interact with its environment, we built a wrapper to map messages from the environment to methods in the translated code. The code inside the DEOS kernel box in Figure 2 is nearly the precise Promela wrapper executed. The code in uppercase indicates code added for environment communication. `START_A_THREAD` starts the thread currently considered running by the scheduler. The exact code is `toThread!resume(Scheduler_itsRunningThread)`. `STOP_A_THREAD` stops a thread that has run for longer than its budget will allow, corresponding to the message `toThread!stop(id)`. Note that `toThread` is a channel on which all messages from the Kernel is sent to all possible threads, hence the `id` needs to be sent along so that the appropriate thread receives the message. `START_THE_TIMER` sends a start message to the timer with the remaining budget of a starting thread using the message `toTimer!start(Timer_time)`.

The three messages that the kernel can receive from a thread, `create`, `delete` and `finishedForPeriod`, correspond directly to DEOS API calls. The other two messages that the kernel can receive, the system tick interrupt and the timer interrupt, correspond to the interrupt handler methods of the `Scheduler` class. The figure does not contain the code for `getTimeRemaining` and its corresponding return message, because they are called by the kernel from within several methods.

Within the DEOS Kernel, the first code to execute is `coldStartKernel()`, which initializes the kernel data structures. Then, the process for the idle thread is started and the timer is started with the idle thread's budget. In our model, all threads (except idle) are created dynamically with an API call to `createThread`. The main thread must be created first because, in DEOS, all threads are allocated time from the main thread in a process' budget.

The rest of the code for the DEOS kernel follows the typical structure of a reactive system: it sits in a loop and reacts to messages it receives from its environment. When the `create` message is received, the `createThread` API call is made which returns the `id` of a thread (the index of the `Thread` object that is created for the thread). The id is used by the thread process to distinguish messages sent and received by it. When a user thread is created its budget is taken from that of the main thread.

When a thread decides to terminate it sends the `delete` message to the kernel. In DEOS, only the currently running thread can delete itself, so a new thread must be scheduled as part of the `deleteThread` call. A thread can yield the CPU before its budget has run out by sending the `finishedForPeriod` message. The kernel then performs the `waitUntilNextPeriod` call for the thread, which also involves scheduling a new thread to execute.

Whenever a `systemTickInterrupt` message is received, a thread of higher priority than the currently executing thread may become schedulable. Therefore, DEOS records the id of the currently running thread (`old`) and checks whether a new thread has been scheduled during the call to `handleSystemTickInterrupt`. If this is the case, a `stop` message is sent to the `old` thread and start the newly scheduled thread and start the timer. Note that if a thread was preempted it is important to find out how much time is still remaining from its budget, since it might get another chance within the current period. This is achieved by sending a `getTimeRemaining` message to the timer, which returns the value in a reply message.

A `timerInterrupt` message indicates that a thread has exceeded its budget and must be stopped immediately. The call to `handleTimerInterrupt` stops a thread and

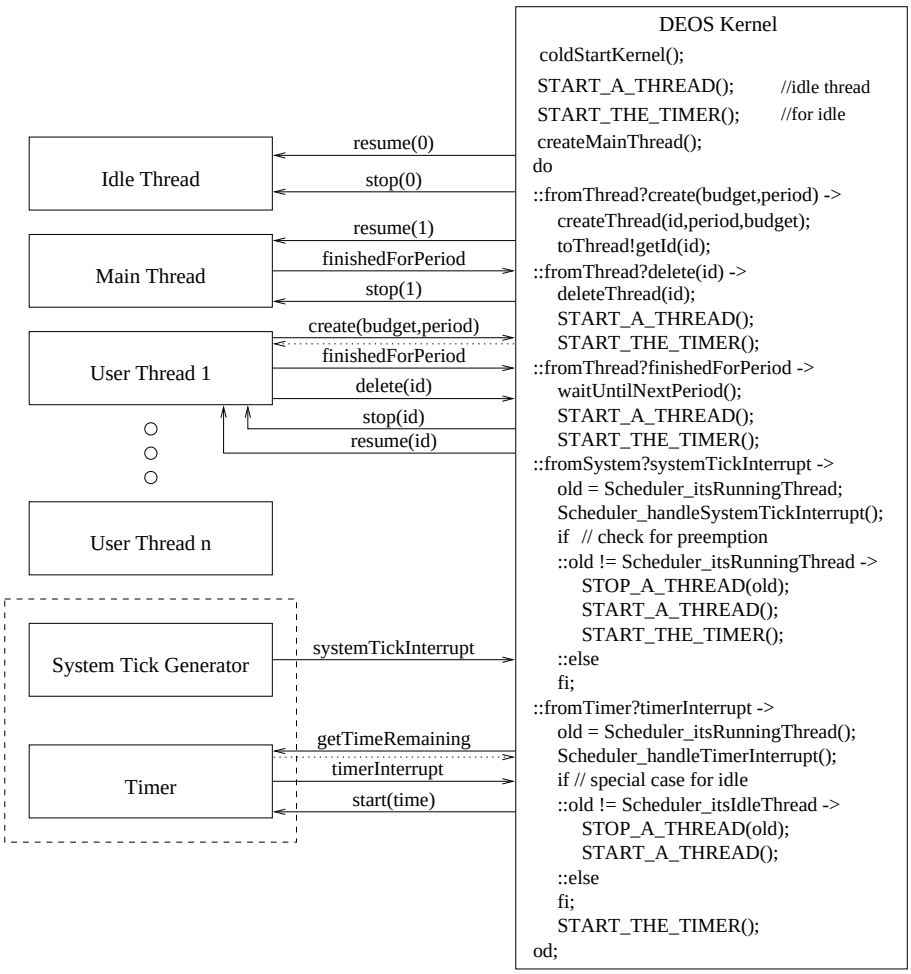

Figure 2: DEOS Kernel and its Environment

schedules a new one. However, when the idle thread is the only thread in the system then it cannot be stopped (since there will be nothing else to run), so the timer is only restarted in this case.

Threads

We distinguish three types of threads: the idle thread, the main thread and user threads. The `User` threads have most functionality: they can be stopped, yield the CPU and decide to terminate. The Promela code for the user threads is shown in Figure 3. A non-deterministic if statement is used to provide an environment where all possible thread behaviors are examined. Only the `UserThread` can make `create` calls, because the other threads are created during initialization. The idle thread can only receive a message to `stop` while the main thread can also decide to yield the CPU. We use synchronization to ensure that when the kernel sends a `resume(id)` message only the thread with the corresponding id will receive it.

```
proctype UserThread(chan fromScheduler, toScheduler;
                    byte myBudget, periodIndex)
{
   byte id;

   byte threadState = threadStatusNotCreated;
   toScheduler!create(myBudget,periodIndex);
   fromScheduler?getId(id);
   threadState = threadStatusDormant;

   do
   ::fromScheduler?resume(eval(id)) ->
        threadState = threadStatusActive;
        if
        ::fromScheduler?stop(eval(id));
        ::toScheduler!finishedforperiod,0,0;
        ::toScheduler!delete,id,0 -> goto terminate;
        fi;
        threadState = threadStatusDormant;
   od;
terminate: skip;
}
```

Figure 3: The DEOS User Thread Model.

Interrupts
Modeling the generation of interrupts was the most difficult part of constructing the environment for DEOS. Promela and Spin do not handle real-time, so the passing of time had to be modeled explicitly. When time must be modeled without tool or language support, the challenge is to determine the level of abstraction at which real-time is modeled.

We started with a simplistic view of both the system tick and the timer interrupts, by creating two processes that could generate interrupts at any time. To determine how long a thread had executed, we did not maintain a timer, but simply picked a nondeterministic value between 0 and the thread's available budget. While this environment was suitable for simulation, it did not take long to realize it was not suitable for verification; When checking properties, we constantly found counter-examples due to some "impossible" behavior by the interrupts. For example, system tick interrupts would occur several times, each indicating that 20 time units had passed, but the timer, set for 10 time units, would never go off. This lack of coordination between the time-related interrupts made it impossible to verify the time partitioning features of the kernel.

To allow the necessary level of coordination, we combined the `SystemTickGenerator` and `Timer` into one process. The timer model keeps track of the time that has been used in a period and makes sure that a system tick interrupt only occurs when the appropriate amount of time has been used (and vice versa).

To limit the number of potential execution paths and avoid state space explosion, we limited the choices as to the amount of time that a thread could execute. In cases where the interrupts do not constrain the amount of time that has passed during thread execution, the timer *nondeterministically* chooses how much time a thread uses. It chooses from three possibilities: either it used no time, or it used all of its time (or all of the time left in the period, if that is smaller), or it used half of the time between the current time and the end of the period. We picked these cases based on intuition similar to that used in selecting boundary cases during testing, with the middle value included for good measure. We conducted several experiments that varied this abstraction and found that the middle value increased the state space by approximately twofold, but did not effect the validity of the verification results.

5 VERIFICATION

Verification in Spin involves systematic execution of all possible process interleavings in a program. It supports assertion violation detection, deadlock detection and model checking of linear temporal logic (LTL) formulae. The main aspect of DEOS that we were interested in verifying was the temporal partitioning property: that each thread in the kernel is guaranteed to have access to its complete CPU budget during each scheduling period. In order to verify this desired property is an actual property of the program, it is necessary to describe the property in terms of the program. For some properties this is non-trivial, because the parts of the program that impact the property may be numerous and difficult to identify. Therefore, specifying these properties can be difficult for people other than the original developers, such as code reviewers or independent verification teams.

We tried two approaches to analyzing the time partitioning properties in the DEOS kernel. The first was to place assertions in the code to identify potential errors. If the model checker finds an assertion violation, the reported error trace can be simulated and it can be determined whether or not the trace is really an error. If it is not an error (i.e. the assertion is too strong) then the assertion can be altered. Additionally, if exhaustive verification is possible, the absence of the assertion violation can be verified with respect to some environment.

A second approach to verifying time partitioning in the kernel is through the use of liveness properties. A liveness property states that some event (or sequence of events) will *eventually* occur in the system. Within Spin, verification of liveness properties is supported using linear temporal logic (LTL). To state a liveness property in terms of program events, labels are placed into the code and referred to from within an LTL property specification.

Assertion Verification: Remaining Budget
In the first verification experiment, we conjectured that any value assigned to a thread's remaining budget should be smaller than the total budget for the thread; otherwise the thread would have access to use too much CPU time. Therefore, we placed an assertion into the code where this value is assigned, in the `setRemainingBudgetInUsec` method of the `Budget` class.

We attempted to verify this assertion with the simplest system configuration, one with only one user thread in addition to the main thread and without dynamic thread creation and deletion. With this configuration, Spin was able to exhaustively verify that the assertion is not violated. This verification searched 8.8 million states with a maximum depth of 199628, requiring 322MB of memory and just over 10 minutes on a Sparc Ultra60.

With dynamic thread creation turned on, Spin found a violation of the assertion in less than a second. It reports a scenario where the main thread (Thread 1) begins executing and the timer is set to 20. Then, a user thread gets created, changing the main thread's budget from 20 to 13. Next, the main thread yields by calling `finishedForPeriod`. The kernel reads the remaining time from the timer (20) and sets this as the main thread's remaining budget. The next step is the assertion, which is violated because the remaining time (20) is more than the main thread's budget (13). This specific error trace is slightly suspect, because the main thread did not use any time. However, any amount of time usage less than

7 would lead to the same assertion violation. Therefore, this execution trace seems to be a valid example of a potential system execution.

The real question is whether this is a valid example of the *intended* system execution. It was not obvious (to the NASA team doing the verification) whether the violation of this assertion would necessarily lead to a violation of time partitioning. If both the main thread and the user thread attempt to execute in the period where the user thread is created, then the CPU budgeting allocation will total over 100%. However, this cannot happen because of a design constraint that the main thread must have the shortest period of all threads. When the user thread is created and becomes ready, it sits waiting for the start of its next period. The main thread, having a equal or shorter period, will have reached the end of its period by this time, and its remaining budget will get reset to the reduced value (13). Therefore, both threads cannot execute in the current period, and the violation of this assertion is safe with respect to time partitioning.

The interesting outcome of this part of the experiment is that it indicates precisely how maintenance of time partitioning by the kernel depends on the main thread running in the shortest period of all threads in a process. To verify that this dependency exists, we changed the code so that the user thread that gets created is in a shorter period than the main thread, and observed that time partitioning was violated. To do this, an assertion was placed in the `startTimer` method of `Budget` to detect when a thread is started while its remaining budget is greater than its total budget. This assertion was used to find situations where the remaining budgets of schedulable threads were greater than 100% CPU utilization, meaning that not every thread could run, and thus violating time partitioning.

Liveness Properties: Idle Execution

If time partitioning is maintained in the system, then we believed that the following liveness property would hold: if there is slack in the system (i.e. the main thread does not have 100% CPU utilization) then the idle thread should run during every longest period[4].

To specify this property, labels were placed in the program to identify when the idle thread starts running and where the longest period begins and ends. The property is then specified as:

```
[]( beginperiod -> (!endperiod U idle))
```

This says it is always (`[]`) the case that, when the longest period begins, it will not end until (`U`) the idle thread runs. That is, idle will always run between the begin and end of the longest period.

[4] Note, this is a necessary condition of time partitioning, and is not sufficient to guarantee time partitioning.

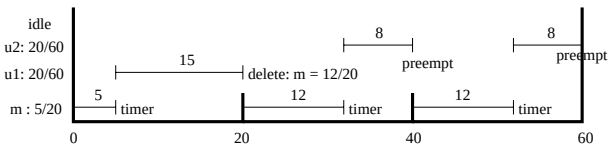

Figure 4: Error scenario

Spin automatically generates a finite state automaton that monitors the system for violations of the LTL property. Verification is done over the combination of the property automaton and the system model. This causes a potential increase of the state space by a factor of 4 in this example, because the property monitor has 4 states. In practice, the increase is approximately 2 fold, because not all of the states are reachable.

One pragmatic difficulty encountered with this verification was that, in order to make the event labels globally visible, it was necessary to remove a large number of `atomic` constructs from the model[5]. This increases the number of process interleavings that Spin must consider, making the verification less efficient.

Verification was run for several system configurations. With 2 user threads and dynamic thread creation and deletion enabled, Spin reported the error scenario partially shown in Figure 4. In this configuration, the main thread runs in period 0 with an initial budget of 19/20. Two user threads are created to run in period 1 with budgets of 20/60. For each user thread, 7/20 is taken from the main thread, leaving it with a budget of 5/20. The total budget in this configuration is 55/60, leaving 5 units for the idle thread to fill at the end of period 1.

Figure 4 shows a scheduling sequence where user thread 1 deletes itself (before being interrupted) at the end of the first period 0. At this point, its budget (20/60 or 7/20) is given back to the main thread, giving it 12/20. The scheduling then continues normally to the end of the period 1 boundary. At this point, Spin signals an error because the idle thread did not run between the two period one boundaries. Notice that user thread 2 only ran for 16 (8+8) and not the 20 it requested, so time partitioning was violated. The error stems from the fact that when user thread 1 deleted itself, it immediately returned its budget to the main thread. This leaves the main thread with a remaining budget of 24 (12+12) and user thread 2 with 20, with only 40 left in period 1. The result is that user thread 2 does not get all of the CPU time it requested.

This bug was, in fact, the same bug that Honeywell had dis-

[5] The Promela `atomic` construct is used to group statements together to prevent interleaving. It was used to model DEOS critical sections where interrupts are disabled

covered during code inspections. Therefore, it would seem that model checking can provide a systematic and automated method for discovering errors. However, it is unclear as to how "lucky" we were in configuring a model that allowed the bug to be discovered. In addition, with dynamic thread creation and deletion enabled, the state space was too large to be exhaustively verified. Therefore, at this point it was not apparent that the model could be searched exhaustively to a depth necessary to guarantee that the error was discovered. In addition, after adding the fix to the code, we were not able to perform exhaustive verification [6]. The following section describes how abstraction was used to guarantee that the error would be discovered and to permit exhaustive verification of the fix.

6 PROGRAM ABSTRACTION

During the initial process of verification, the state space of the DEOS model was very large if dynamic thread creation and deletion was enabled. Experiments using Spin in supertrace mode suggested that the number of states in the system was on the order of 20 million states, which was 2-4 times too large to handle within the 512MB of available RAM. Therefore, we needed a way to reduce the state space by this small factor.

The first step in abstracting a program is to identify some part of the program as a good target for abstraction. The second step is to introduce the abstraction. We describe below how we used information about the program to identify a part of the program to be abstracted. We believe that the program pattern that we applied abstraction to is widely used, and therefore this abstraction may be useful in many other applications. We then introduce the abstraction using *predicate abstraction*, a formal technique based on abstract interpretation.

In identifying part of the program to abstract, we were guided by several experiments showing traces through the system that were 2,000,000 steps long. Intuitively, this seemed too large considering that the system's behavior is cyclic in nature: at the end of the longest scheduling period, the system should return to a state where all threads are available to be scheduled with all of their budget available. These long traces indicated that some data was being carried over these longest period boundaries. We were able to identify this data by running a simulation and observing the Spin data values panel; The `itsPeriodId` data member for the `StartOfPeriodEvent` class was operating as a counter, incrementing every time the end of the corresponding period was reached. In addition, the `itsLastExecution` variable in the Thread class was also climbing, because it is periodically assigned the value of the `itsPeriodId` counter for the `StartOfPeriodEvent` corresponding to the thread's scheduling period.

[6] The fix involves keeping track of the budget of deleted threads and returning them to the main thread at the end of the deleted thread's period.

```
void StartOfPeriodEvent::pulseEvent() {
  countDown = countDown - 1;
  if (countDown == 0) {
    itsPeriodId = itsPeriodId + 1;
  ...
}

void Thread::startChargingCPUTime() {
  // Cache current period for multiple uses here.
  periodIdentification cp;
  cp = itsPeriodicEvent->currentPeriod();
  ...
  // Has this thread run in this period?
  if (cp == itsLastExecution) {
    // Not a new period. Use whatever budget is remaining.
    ...
  }
  else {
    // New period, get fresh budgets.
    ...
    // Record that we have run in this period.
    itsLastExecution = cp;
    ...
  }
  ...
}
```

Figure 5: Slice for `itsPeriodId`

A slice of DEOS with respect to `itsPeriodId` and `itsLastExection` is shown in Figure 5. These variables are used to determine whether or not a thread has executed in the current period; If it has not, then its budget can be safely reset. `itsLastExecution` is assigned the value of `itsPeriodId` (the return value of `itsPeriodicEvent->currentPeriod()`) whenever the two are not equal.

The information that actually needs to be maintained is simply a boolean variable that indicates whether a thread has executed in the current period. These flags would then be reset at every period boundary. However, this approach can not be implemented in the system for efficiency reasons: all kernel algorithms must be O(1), where as resetting the flags is O(n), where n is the number of threads.

This realization led us to try a technique called predicate abstraction, where program variables are replaced by a predicate (or set of predicates) that describe some relation over the variables. In this case, we replaced the variables `itsPeriodId` and `itsLastExecution` by a single boolean variable, `executedThisPeriod`, defined by the predicate `itsPeriodId` == `itsLastExecution`.

To generate an abstract program, the statements that manipulate the variables must map to statements that properly update the predicate variable. In this case, it is obvious that the statement `itsLastExecution = itsPeriodId` should be mapped to `executedThisPeriod = TRUE`. However, the mapping for the statement `itsPeriodId = itsPeriodId + 1` is nontrivial because, depending on the previous values of `itsPeriodId` and `itsLastExecution`, the value of the predicate after the

Concrete Program	Abstract Program
int itsPeriodId; int itsLastExecution;	bool executedThisPeriod;
itsPeriodId = itsPeriodId + 1;	executedThisPeriod = FALSE;
itsLastExecution = itsPeriodId;	executedThisPeriod = TRUE;

Table 1: Abstraction of `itsPeriodId` and `itsLastExectution` to a single boolean

increment could be either `TRUE` or `FALSE`. However, in the real system, `itsPeriodId` is always incremented, and `itsLastExecution` is only ever assigned the value of `itsPeriodId`. Therefore, it is easy to prove (by inspection of the code in Figure 5) that `itsPeriodId` will always be greater than or equal to `itsLastExecution` and therefore the result of incrementing `itsPeriodId` will be that the predicate becomes `FALSE`. This abstraction mapping is shown in Table 1.

In practice, the case where `itsPeriodId` rolls over (at MAXINT) is an exception to the above assumption. However, the correct behavior of the real system also depends on this assumption (specifically, that `itsPeriodId` does not roll over and catch up with `itsLastExecution`, meaning that a thread will not wait MAXINT periods). This is precisely the case where the above predicate abstraction will become invalid. Therefore, this abstraction does not introduced any stronger assumptions on the system than those imposed by the implementation, meaning it is sound.

The actual abstraction used is slightly more complex than the mapping in Table 1 because there is a one-to-many relationship between `StartOfPeriodEvents` and `Threads`. Therefore, when `itsPeriodId` is incremented, a predicate must be updated for *every* thread in the period. This corresponds exactly to the O(n) updating algorithm that could not be used in the implementation. However, the O(1) real-time constraint does not apply to the verification model.

With this abstraction in place, there was a reduction in the size of the state space by several orders of magnitude. As a result, we were able to guarantee that we could find the liveness property violation and perform exhaustive verification within 512MB after adding the fix.

7 RELATED WORK

Although model checking is becoming popular for the analysis of software specifications and designs [1, 3, 7], it is not commonly used for analyzing implementations. Holzmann and Smith developed a system whereby a Promela model is constructed directly from stylized C code [15]. There technique differs from ours since abstraction is done during translation, whereas we abstract the source before translation. An approach closer to our own is used within the Java PathFinder tool that automatically translates Java programs to Promela [13]. An alternative is to extend the expressibility of model checking languages with programming language features [9, 21].

The technique of predicate abstraction was developed by Graf and Saidi [11] and extended by others [8, 20]. We use predicate abstraction in conjunction with techniques developed by Hatcliff et al. that use abstraction and specialization techniques to reduce Java programs for verification [12].

Dwyer and Pasareanu are developing automatic ways of generating environments for software systems to allow efficient model checking [10, 17]. Although we experimented with their filtering techniques, we found that the timer model constraints were more easily expressed operationally in Promela rather than declaratively as LTL constraints. However, further study is warranted because this is a critical problem.

8 CONCLUSIONS AND FUTURE WORK

The results of this experiment indicate that it could be highly beneficial to continue to pursue the research and development necessary to provide model checking support directly for programming languages, and to explore the use of model checking in the formal design and code review processes. We believe that the fact that model checking could be applied effectively to the source code implementation of the DEOS scheduler represents one of the most significant applications of model checking to date[7]. We used abstraction in a controlled and minimal fashion, which helped to provide an understanding of exactly what must can be done to avoid state space explosion. However, further research is still needed to increase the ease of automated translation and for automated support for program abstractions.

The most difficult part of defining the environment of DEOS was to develop a model for the interrupt generation that would allow us to check the properties of interest. We believe that if this problem is not addressed, it will seriously hamper the adoption of model checking as a tool to find errors in programs. One solution might be for a programmer to develop an abstract environment in parallel with the system. However, this is a paradigm shift from the current approach of plugging the system into the real environment for testing. We believe that a closer integration of model checking and testing may aid in the adoption of the technology.

[7] The majority of our research colleagues to whom this project was mentioned expressed, at a minimum, high levels of skepticism that this problem was tractable.

We are continuing to extend the model of the DEOS kernel with additional features. We have recently extended the model to include a new feature of DEOS that allows threads to request slack time from the system. One effect of this change is that it is no longer necessary for the idle thread to execute in every longest period. Therefore, we develop a new formulation of the time partitioning property in terms of an invariant, which we believe is a sufficient condition for the maintenance of time partitioning.

9 ACKNOWLEDGMENTS

We thank Klaus Havelund, SeungJoon Park, Charles Pecheur, Michael Lowry, Thomas Uribe, Hassen Saidi, Matt Dwyer, John Hatcliff and David Dill for numerous technical discussions that contributed to this work.

REFERENCES

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 of *SIGSOFT Software Engineering Notes*, pages 156–166. ACM, October 1996.

[2] Pam Binns. Design document for slack scheduling in deos, draft alpha.3. Honeywell, September 1998.

[3] W. Chan, R. Andersen, P. Beame, D. Jones, D. Notkin, and W. Warner. Decoupling Synchronization from Local control for Efficient Symbolic Model Checking of Statecharts. In *ICSE 21*, pages 142–151, Los Angeles, May 1999.

[4] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite-State Concurrent Systems. In *A Decade of Concurrency: Reflections and Perspectives*, Lecture Notes in Computer Science 803, 1993.

[5] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Translactions on Program Languages and Systems*, 16(4), sep 1994.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] Z. Dang and R. Kemmerer. Using the ASTRAL Model Checker to Analyze Mobile IP. In *Proceedings of the 21st International Conference on Software Engineering*, pages 132–141, Los Angeles, May 1999.

[8] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of CAV'99*, pages 160–171, 1999. Lecture Notes in Computer Science 1633.

[9] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 6th SPIN Workshop*, Lecture Notes in Computer Science 1680, 1999.

[10] M. Dwyer and C. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM SIGSOFT, November 1998.

[11] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Vericifaction*, Lecture Notes in Computer Science 1254, pages 72–83, 1997.

[12] J. Hatcliff, M. Dwyer, S. Laubach, and D. Schmidt. Staging static analyses using abstraction-based program specialization. In *LNCS 1490 Principles of Declarative Programming: 10th International Symposium*, sep 1998.

[13] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 1999.

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

[15] G. Holzmann and M. Smith. Software model checking - Extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497. Kluwer, October 1999.

[16] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Translation and verification of the DEOS scheduling kernel. Technical report, NASA Ames Research Center / Honeywell Technology Center, October 1999.

[17] C. Păsăreanu, M. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of the 6th SPIN Workshop*, Lecture Notes in Computer Science 1680, 1999.

[18] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, Lecture Notes in Computer Science 137, 1982.

[19] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-178B, RTCA, Inc., dec 1992.

[20] T. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, April 1999.

[21] W. Visser, K. Havelund, and J. Penix. Adding Active Objects to SPIN. In *Proceedings of the 5th SPIN Workshop*, Trento, Italy, July 1999.