Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report*

Vu Ha, Murali Rangarajan, and Darren Cofer†
Honeywell International
3660 Technology Drive
Minneapolis, MN 55418

Harald Rueß and Bruno Dutertre‡
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025

Abstract

The hardware and software in modern aircraft control systems are good candidates for verification using formal methods: they are complex, safety-critical, and challenge the capabilities of test-based verification strategies. We have previously reported on our use of model checking to verify the time partitioning property of the Deos™ real-time operating system for embedded avionics. The size and complexity of this system have limited us to analyzing only one configuration at a time. To overcome this limit and generalize our analysis to arbitrary configurations we have turned to theorem proving.

This paper describes our use of the PVS theorem prover to analyze the Deos scheduler. In addition to our inductive proof of the time partitioning invariant, we present a feature-based technique for modeling state-transition systems and formulating inductive invariants. This technique facilitates an incremental approach to theorem proving that scales well to models of increasing complexity, and has the potential to be applicable to a wide range of problems.

1 Introduction

Integrated modular avionics (IMA) architectures found in modern aircraft contain applications of different criticalities executing on the same CPU. The execution of these applications must be scheduled so that they do not inadvertently consume CPU time that has been budgeted for other applications. This scheduling function may be performed by a real-time operating system (RTOS) that enforces *time partitioning* between scheduled tasks. The complexity of this embedded software and its criticality for safe operation present many challenges to the test-based verification technology currently in use.

Scheduling has been perhaps the most widely researched topic within real-time systems, and should provide a sound foundation to the development and validation of real-time systems. Even though many types of scheduling disciplines and different task sets have been investigated theoretically, real-life schedulers are often richer and more complex than that considered in the literature, and thus there is usually a gap between the abstract models used to derive scheduling results and the real scheduling algorithms used in an RTOS. The challenge now is to obtain high assurance that such general methods have been properly applied to the given system.

Over the past several years, Honeywell has been developing and applying formal techniques to verify safety-critical properties in IMA software components. The focus of much of our work has been the scheduler of the Deos real-time operating system, both as a test case for a variety of tools and techniques, and as an important and "industrial strength" verification problem in its own right.

This work and the approach taken began with an earlier project undertaken with the Automated Software Engineering group at NASA Ames [13]. The model checker Spin [9] was used to model and verify a portion of an early version of the Deos scheduler. The verification model was derived directly from the actual flight code so that the analysis results could be traced back to the real system.

More recently, we have focused on the modeling and analysis of advanced features of the latest version of Deos, such as slack scheduling, aperiodic interrupt servicing, and mutex handling [4]. These new features add considerably to the complexity of the software. We have also analyzed the correctness of the overhead time calculations performed in the scheduler [5]. Formal analysis of the system has allowed us to examine its behavior over a wider range of scenarios than is possible by testing alone and increased our confidence in the correct operation of the system.

As we have continued to add features to our model and correspondingly expanded the size of its state space, we have reached the memory limits (and practically speak-

*This material is based upon work supported in part by NASA under cooperative agreement NCC-1-399.

†{vu.ha,murali.rangarajan,darren.cofer}@honeywell.com
‡ruess@csl.sri.com,bruno@sdl.sri.com

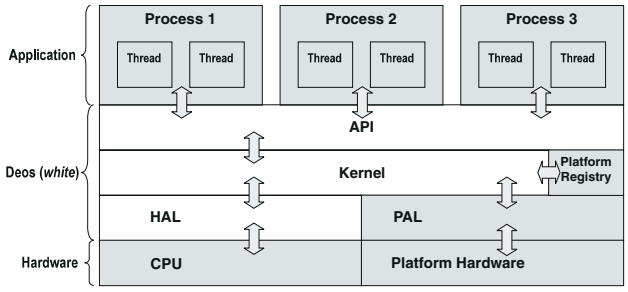


Figure 1. Deos components and terminology

ing, the time limits) of our model checking tools. Thus, for the larger systems we have considered, model checking serves as an effective debugging aid but is not able to provide an exhaustive verification. In addition, model checking requires that a specific system configuration be modeled and analyzed. In our case this means fixing the number of threads, the number of periods, and the thread budgets.

The goal of our current work is to perform a comprehensive analysis of time partitioning in Deos for arbitrary configurations. We have used the PVS theorem prover to model the operations of the scheduler and reason about its temporal properties. This naturally means modeling at a higher level of abstraction than in the previous model checking approach, but our results extend to any legitimate initial specification of threads, periods, and budgets.

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 is certified to DO-178B [15] Level A, the highest possible level of safety-critical certification.

The main components of a Deos-based system are illustrated in Figure 1. A given software application consists of one or more processes. Each process is executed as one or more 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 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). The Deos scheduler supports a number of features such as aperiodic interrupt threads, synchronization mechanisms such as mutexes and semaphores, and communication primitives such as periodic IPC and mailboxes. Our current model consists only of those features that have the most impact on time partitioning, such as RMS scheduling, thread creation/deletion and slack recovery.

2.1 Scheduler Operation

In Deos, each thread is given a periodic time budget and has a fixed priority. To maximize utilization, Deos requires the thread periods to be *harmonic*: If two threads have distinct periods P_1 and P_2, then one of the periods must be a multiple of the other.

The Deos scheduler is based on a *rate monotonic scheduling* (RMS) policy. RMS assigns thread priorities so that threads of short periods (high rates) are assigned higher priorities than threads of long periods (low rates). 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.

Deos fully supports *dynamic thread creation and deletion*. At any given time, the running thread can *create* a new thread that becomes eligible to run at the next period boundary for its period. Also, the running thread may *delete itself*, after which the scheduler will activate another thread. Intuitively, at thread deletion, the scheduler should "reclaim" the deleted thread's periodic budget and make it available for future thread creation. At thread creation, the scheduler should ensure that the budget of the new thread, in addition to all the existing threads, does not violate time partitioning. Because threads belong to harmonic periods of different lengths, these computations are non-trivial, and realized in Deos by means of a data structure called *budget updated vector*, or *buvec*.

It is often suboptimal to restrict the applications to the use of a fixed CPU time budget in each period. Many applications such as network service applications have highly

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

variable demands on the CPU and have a desired level of performance considerably greater than the minimum required. Giving these applications only the minimum budget necessary will result in low performance; giving them a high enough budget to ensure the performance desired will result in severe underutilization of the CPU most of the time and may *crowd out* other applications that users want to have in the system. For this reason, the Deos kernel provides a mechanism for slack scheduling that assigns *system slack time* to threads on a first-come first-served basis. The classical view of slack scheduling is given in [10]. The version implemented in Deos incorporates several major modifications of this view necessitated by the special features of Deos (e.g. the capability to dynamically activate and deactivate threads, and the existence of aperiodic interrupt threads) [2].

2.2 Example Execution Timeline

Deos's scheduling algorithm is best understood through an example of the system execution timeline. We will use this example as a running example throughout the paper. Presently, our model does not include scheduler overhead. This means that operations such as thread creation or context switch happen instantaneously. A transition in this model may represent a simple change such as the current thread spends one time unit, or may capture complex changes such as new thread creation and/or context switch. At period boundaries, or SOP (start of period) in Deos's terminology, transitions are responsible for refreshing the queue of waiting threads, as well as other bookkeeping tasks such as those related to available budget and slack. Figure 2(a) shows the various states individual threads may exhibit.

The example scenario consists of two periods P_1 and P_2 of length 10*ms* and 20*ms* respectively [1]. Note that the length of the longer period is an integral multiple of (harmonic to) the shorter period. At startup, there are two threads — the main thread (M) with a budget of 5 running in period P_1, and a user thread U_1 with a budget of 6 running in period P_2. U_1's budget can be normalized to the length of the fastest period (P_1) as $6/(20/10) = 3$. The total normalized budget is thus $5+3 = 8$. The remaining normalized time of 2 in the fastest period is not scheduled for any thread, and is known as *time-line slack* or *associated budget*. In addition, a special system thread, the *idle thread*, is automatically created at system start-up. The idle thread (I) runs at the slowest period, has the lowest priority of all threads, is available to run at all times, and is scheduled to run when no other thread is eligible to run. It is special in the sense that it can never complete early, delete itself or spawn other threads.

[1] From now on, all times are implicitly stated in *ms*.

Figure 2(b) shows how the timeline unfolds over four P_1's (40*ms*). Since M has the shortest period, it has the highest priority and is scheduled to run first. It spends its budget (5), and then is *interrupted by the timer*. U_1 is scheduled to run next, and after running for 3, U_1 *deletes itself*. Since no threads are eligible to run, the idle thread I is activated and runs until the end of P_1 (time = 10). At this point, the start-of-period (SOP) updates take place. These include refreshing the budgets of all the threads belonging to all the periods that end at this time, re-setting all slack reclaimed at all periods ending at this time, and other housekeeping tasks.

Since M now has its budget replenished, it is selected to run. At time 12, M *creates* user thread U_2, which runs in P_2 with a budget of 8. This results in the timeline slack becoming $10-5-8/2 = 1$. Unlike M and U_1, thread U_2 is *slack-enabled*, meaning that it may request to run on slack time. U_2 has to wait until the start of its next period (20) before it becomes eligible to run. M continues until it runs out of time (15), at which point it is replaced by I.

At time 20, both M (with replenished budget) and U_2 become eligible to run. M runs first, since it has the shorter period. It *completes early* at time 23 and gives up $5-3 = 2$ to slack. U_2 is now scheduled to run until 30.

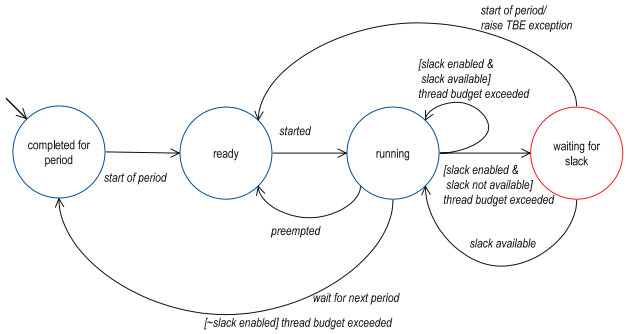
At time 30, M is again eligible to run. Since M has higher priority, it *preempts* U_2. M runs until time 32, at which point it completes early and gives up $5-2 = 3$ to slack. U_2 then gets *reactivated* and continues to run until it spends all of its budget at time 33 ($32+(8-(30-23))$). Since U_2 is slack-enabled, it requests slack time, and receives 4. This slack consists of 3 given up by M at time 32 and 1 of timeline slack. The 2*ms* given up by M in the previous period (at time 23) are lost since it was not used in that period. At time 37, U_2 is interrupted by the timer, at which point I takes over.

To prove time partitioning using PVS, we started by a simple scheduling model, and incrementally added features to it. In our basic model a fixed set of threads execute at periodic rates and may either run until their budget is exhausted or they complete early. It is almost obvious that the time partitioning property will hold in such a simple system, but its inductive proof presented some challenges. To this basic model we subsequently added new features: 1) dynamic threads, which can be created or deleted at any time step, and 2) slack time reclamation, in which threads can run beyond their budgeted time if there is slack in the schedule. The inductive proof was constructed in parallel, following an incremental approach.

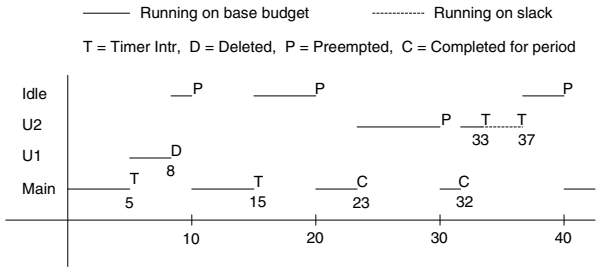
3 Time Partitioning in the Basic Model

We model the execution timeline of Deos using a discrete time state-transition system. There are two central notions

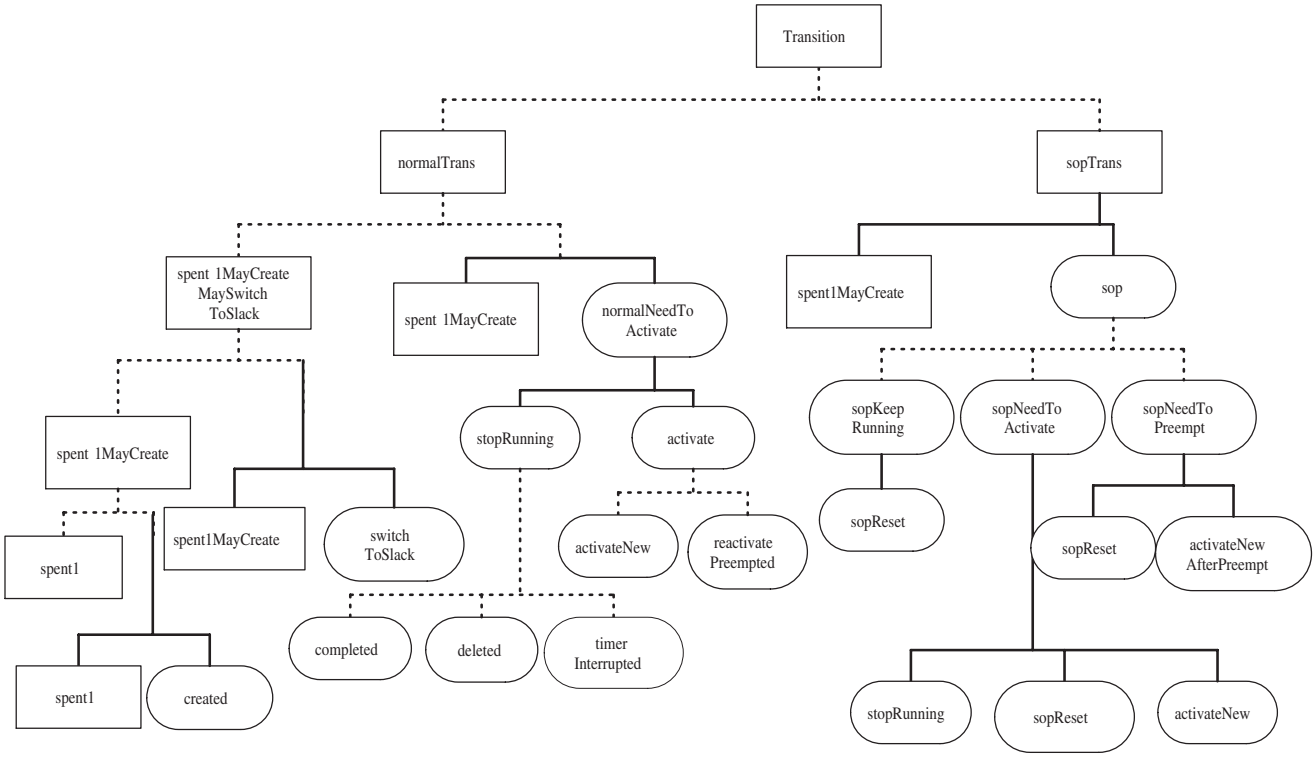
Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**



(a) Diagram of threads states



(b) An example of the Deos scheduler execution timeline



(c) Transition Hierarchy. A transition is either a normal transition or an SOP transition. This "OR" relation is depicted using a dotted line. An SOP transition is defined as a sequence of two sub-transitions: `spent1MayCreate` and `sop`. This "AND" decomposition is depicted using a solid line.

Figure 2. Example execution timeline, thread states, and PVS transition hierarchy of the Deos scheduler.

in this model: *state* and *transition*. At any given time `t` (`t` is a natural number), Deos is in a state `s(t)`. The state `s(0)` is the initial state of Deos at cold start. Deos progresses from state `s(t)` to state `s(t+1)` by means of transitions. Transitions have preconditions that render them applicable only in certain states. Thus we can think of the Deos execution timeline as if generated by a nondeterministic automaton. Our first model describes the basic priority-based scheduling, without dynamic threads or slack recovery.

3.1 System State

The type `state` is defined as a *record type* in which a set of attributes describes relevant features of the system. These attributes are: `time`, which is the current time counted from the last cold start; `itsRunningThreadId` is the id of the running thread; `sopReset` is a boolean attribute that is true only at period boundaries, after all SOP-related updates have been completed; `threads` is *the set of threads*; `itsAssociatedBudget` is the *associated budget* or *timeline slack*.

```
State: TYPE = [#
  time: Time,
  itsRunningThreadId: ThreadId,
  sopReset: bool,
  threads: Threads,
  itsAssociatedBudget: NormalizedBudgetOrZero #]
```

The definition of states as record type does not have any constraints. *Proper states* must satisfy a number of constraints such as: The thread with id `itsRunningThreadId` is the only thread that is running. These constraints are built into the predicate `properState`:

```
properState(s): bool =
  properThreads(s) AND
  properItsAssociatedBudget(s) AND
  properSopReset(s)
ProperState: TYPE = {s: State | properState(s)}
```

The *initial state*, which is the state at cold start, is defined using the following predicate:

```
init(s): bool =
1.  s`time=0 AND
2.  threadsInitialized(s`threads) AND
3.  s`itsRunningThreadId=mainThreadId AND
4.  s`itsAssociatedBudget=
    L0-totalNormalizedBudget(s) AND
5.  s`sopReset
```

A state is an initial state if `time` is zero (condition 1), `threads` are initialized (conditions 2), the main thread is running (condition 3), `itsAssociatedBudget` is equal to the available timeline slack (condition 4, here `L0` is the length of the fastest period), and `sopReset` is true (condition 5).

Reachable states are either initial states, or states that can be reached by transitions from other reachable states. Thus we are adopting an approach widely used in modeling state-transition systems.

```
reachable(s:State): INDUCTIVE bool =
  init(s) OR
  ((EXISTS (ps:ProperState) :
    transition(ps,s) AND
    reachable(ps)))
ReachableState: TYPE={s|reachable(s)}
```

3.2 Transitions

We define *transitions* by means of binary relations: a transition *from a proper state* `ps` to a state `s` is captured by the binary predicate `transition(ps, s)`. Transitions have many types, organized in the *transition hierarchy* (Figure 2(c)). We put a constraint on all types of transition, requiring that they advance the system's clock by one time unit. This time constraint is necessary in order to prove inductive properties about reachable states. Note also that a transition is defined on an ordered pair of proper state (`ps`) and state (`s`). Thus we will need to make sure that all transitions are closed on the set of proper states:

```
transition_proper: LEMMA
  transition(ps, s) IMPLIES properState(s)
```

A transition is either a normal transition or an SOP transition, depending on whether the destination state of the transition (`s`) is at a start of some period. In the example, transition from time 4 to 5 is a normal transition, while transition from time 9 to 10 is an SOP transition.

Normal Transitions

```
normalTrans(ps,s): bool=
  notSop(s) AND
  (spent1(ps,s) OR
  EXISTS (ps1:ProperState):
    spent1(ps,ps1) AND
    normalNeedToActivate(ps1,s))
```

There are two types of normal transitions, one that does not involve a context switch (`spent1`), and one that does involve a context switch (i.e. a new thread needs to be activated: `spent1(ps, ps1) AND normalNeedToActivate(ps1, s)`). Transition `spent1` (e.g from time 1 to time 2) captures the situations where the current running thread spends one time unit. The sub-transition `normalNeedToActivate` captures the situations where the current running thread completes for period, then a context switch happens: either a new thread is activated (time 5), or a previously preempted thread gets reactivated (time 32). Note that this sub-transition has zero duration.

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

At the leaf-level of the transition hierarchy, we need to define (sub-)transitions in terms of the effects they exercise on the system state. For example, transition `spent1` is defined as:

```
spent1(ps,s): bool=
1. (notSop(ps) OR ps`sopReset) AND
2. (idleIsNotRunning(ps) IMPLIES
     currentRunningThread(ps)`timeSpent+1<=
     totalBudget(currentRunningThread(ps))) AND
3. timeIncremented(ps, s) AND
4. s`itsRunningThreadId=ps`itsRunningThreadId AND
5. threadWithId(s, ps`itsRunningThreadId)=
    currentRunningThread(ps) WITH
    [(timeSpent):=
    currentRunningThread(ps)`timeSpent+1]) AND
6. (FORALL (threadId: ThreadId):
      threadId/=ps`itsRunningThreadId IMPLIES
        threadWithId(s,threadId)=
        threadWithId(ps,threadId)) AND
7. s`sopReset=false AND
8. s`itsAssociatedBudget=ps`itsAssociatedBudget
```

Transition `spent1` is defined by means of 8 conditions that enforce parts of the Deos scheduler's policy. For example, condition 2 makes sure that, except for the idle thread which does not have a budget, the current running thread has not spent its entire budget. If it has, `spent1` should *not* be applicable (a context switch must happen). After `spent1`, the system state stays the same, except for the current running thread, which has its `timeSpent` property incremented (conditions 5 & 6). In order to make sure that this transition is closed on the set of proper states, we need to prove the following lemma:

```
spent1_proper: LEMMA
  spent1(ps, s) IMPLIES properState(s)
```

We now briefly summarize the rest of the leaf-level subtransitions. `completed` (e.g at time 23) describes what happens when a thread completes: its unused time, if any, is lost to the idle thread. `activateNew` (e.g. at time 5) and `reactivatePreempted` (e.g. at time 32) capture the activation of the next running thread. Sub-transitions such as `completed` are *not* closed on the set of proper states: after the current thread stops running, there is no running thread in the system. The system's state will become proper again only after the context switch (`activate`) is completed.

SOP Transitions

SOP transitions are those that happen at period boundaries. An SOP transition is defined as a sequence of `spent1` and `sop`. The sub-transition `sop` captures the changes that take place at period boundaries, and is of any of the following three types:

```
sop(ps, s): bool =
  sopKeepRunning(ps, s) OR
  sopNeedToActivate(ps, s) OR
  sopNeedToPreempt(ps, s)
```

Here `sopKeepRunning` is where the current thread keeps running after the period boundary (not present in the running example); `sopNeedToActivate` is where a non-preemptive context switch happens (not present in the running example); and `sopNeedToPreempt` is where the current thread is preempted by a higher priority thread that has just become runable (e.g. transitions from time 9 to 10 and 19 to 20 where idle is preempted, and transition 29 to 30, where U_2 is preempted). A common step of these three types of subtransitions is transition `sopReset`, which specifies what happens at period boundaries. This is the most complex subtransition in the model.

3.3 Proving Time Partitioning

In state-transition systems, properties such as time partitioning (TP) are formulated as predicates on the set of states, and proved to hold for all *reachable states*. The corresponding PVS proof is inductive. It consists of the base step and the inductive step:

```
init_invariant: LEMMA
  init(s) IMPLIES TP(s)
transition_invariant: LEMMA
  TP(ps) AND transition(ps, s) IMPLIES TP(s)
```

The base step is straightforward; the challenge is to formulate TP so that we can carry out the inductive step. In our case, TP states that at any period boundary for any period P_i, all threads that run in periods not slower than P_i must be satisfied for their current periods. A thread is said to be *satisfied for its current period* if either 1) it has run for its budget during its current period or 2) it has completed early for its current period. We will refer to this formulation as `TP1`. In the running example, at time 30, which is an SOP for P_1 but not P_2, M is the only thread that runs in P_1, and since it completed early at time 23, `TP1` holds.

Formulating TP Invariant: The First Attempts

`TP1` is a more precise version of the description often used informally: "all threads should have access to their periodic budget". It is *verifiable* in model checking approaches. In fact, it is used in our ongoing research that uses the SPIN model checker to verify Deos's time partitioning. But this formulation is clearly unsuitable for the inductive theorem proving approach we are pursuing. We need a formulation that can be checked for every state, not just SOP states. Toward this end, we first attempted to formulate time partitioning based on the concept of *commitment*. Intuitively, at any given state s and any given period P_i, we can define the commitment that the system has made for threads that run in periods not slower than P_i. Time partitioning holds if this commitment does not exceed the time remaining until

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

the next SOP of P_i. Equivalently, we say that the system is not *overcommitted* or has *good commitment*.

```
goodCommitment(s,period): bool=
  commitment(s,period)<=remainTime(s,period)
TP2(s,period):bool =
  goodCommitment(s,period)
TP2(s):bool =
  FORALL (period): TP2(s,period)
```

In the running example, at time 3, the remaining times for P_1 and P_2 are 7 and 17, respectively. For P_1, `commitment(s, 1)` $= 2 < 7$, since M still has 2ms more to run. For P_2, in addition to the commitment of 2 until time 10, the system needs to accommodate M's budget of 5 from time 10 to time 20, as well as U_1's budget of 6, for a total of `commitment(s, 2)` $= 2 + 5 + 6 = 13 < 17$.

Clearly, `TP2` $\Rightarrow$ `TP1`. The reason is that at SOP of P_i, the remaining time is zero, and so the commitment must also be zero, and so all threads of periods not slower than P_i must be satisfied for their current periods. Conversely, if the system is overcommitted at any time with respect to some P_i, then time partitioning may be violated if all threads run to their full budgets until the next SOP of P_i. Thus, `TP1` $\Rightarrow$ `TP2`, and we have successfully formulated time partitioning as a predicate that can be checked at any time.

Unfortunately, using `TP2` it is not possible to inductively prove that the system is never overcommitted, even when we are confident (using other methods) that TP holds. In an inductive proof, we need to rely exclusively on the inductive hypothesis (`TP(ps)` in this case) to deduce the inductive conclusion (`TP(s)`). This deduction breaks down, for example, in the transition from time 8 to time 9. Because idle is running during this time, the commitment stays the same, while the remaining time is (as always) decremented.

Disjunctive TP Invariant

The key to resolving the above problem in our inductive proof lies in the observation that whenever the commitment for period P_i stays the same after a transition, the current running thread must be of a period slower than P_i. This, together with the fact that the scheduler is RMS-based, imply that all threads of period not slower than P_i must be satisfied for their current period. This leads us to adopting a disjunctive invariant form:

```
TP3(s,period): bool =
  goodCommitment(s,period) OR
  FORALL (threadId):
    threadWithId(s,threadId)`period<=period
      IMPLIES satisfied(s,threadId)
```

It is an easy but interesting exercise to check that `TP3` $\Rightarrow$ `TP1`, and since `TP1` $\Leftrightarrow$ `TP2`, we have `TP3` $\Rightarrow$ `TP2`. But since `TP3` is a version of `TP2` weakened by a disjunctive term, `TP3` $\Leftrightarrow$ `TP2`. In short, all three formulations are equivalent, but only `TP3` is amenable to an inductive proof!

After settling with the invariant `TP3`, the inductive proof is straightforward conceptually (albeit still rather laborious, especially for SOP transitions). The entire collection of theories has a total 1648 lines of PVS code and 212 lemmas.

Our proof techniques build upon a systematic method proposed by Rushby [16] for constructing disjunctive invariants when verifying state-transition systems. This approach is related to the proof diagrams proposed by Manna and Pnueli [12]. It has been successfully applied to a variety of systems including fault-tolerant group membership algorithm [14] and security protocols [7].

4 Feature-Based Incremental Proof

Next, we expand the basic scheduler model to include dynamic thread creation and deletion. We first describe the changes to the model, and discuss the efforts involved in proving time partitioning in this enhanced model. We then further expand the model and proof to include slack scheduling.

4.1 Adding Dynamic Threads

Changes to the model

In this model, the current running thread can either create a new thread, or delete itself (except for the idle thread). The `state` record type now has a new field: `buvec`. This is an array of normalized budgets, indexed by the system's periods. It is initialized to an array of zeros. Except for `sopReset`, the existing transitions in the basic model do not have any effect on `buvec`, and as such every lemma about them in the basic model can be carried over to the new model with little effort[2]. More work is required for sopReset as it updates buvec.

Thread creation and deletion are also modeled as subtransitions with zero duration. The most important constraint at thread creation is `enoughBudget`, which ensures that, starting from the the next period boundary of `period`, the newly created thread will be guaranteed of `newThreadBudget` for every `period`. In the running example, M was able to create U_2 (which has normalized budget of 4), because the total normalized budgets of M and U_2 is 9, which is less than `L0`=10. In general, checking `enoughBudget` can be more involved than this. Specifically, if a thread of `period` deletes itself, its budget should be reclaimed for future thread creation. But in the same `period` when the deletion occurs, this reclaimed budget

[2] In our experience, most effort in reproving a lemma involves minor editing of the PVS proof that can be completed in minutes.

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

should *not* be used for creating threads that run in periods faster than `period`. The array `buvec` was introduced precisely to address this issue. When a thread of `period` deletes itself, its normalized budget is added to `buvec[period]`. Then `enoughBudget` is defined as:

```
enoughBudget(newBudget,s,period):bool=
  newThreadBudget<=s`itsAssociatedBudget+
  sigma(0,period,LAMBDA(j:Period): buvec(j))
```

Changes to the Proof

With the addition of dynamic threads, the inductive step in proving `TP3` must be expanded with invariant proofs for the new sub-transitions. The inductive step for the `sopReset` also needs to be updated. Other parts of our theory largely stay the same.

After a thread deletion, the system's commitment decreases by an amount equal to the deleted thread's remaining budget. Since the remaining time stays the same, the system can not *become* overcommitted after thread deletion. The more tricky case is at thread creation, where the system's commitment is increased by the commitment to the new thread, while the remaining time stays the same. Our solution is to modify the definition of `goodCommitment` as follow:

```
goodCommitment(s,period):bool=
  commitment(s,period)+
  buvecCommitment(s,period)<=remainTime(s,period)
```

Notice the new term `buvecCommitment`. It is defined to have the following properties. First, it should be non-negative at all times, and zero at initial states. Second, at thread deletion, it should increase by an amount *at most* equal to the deleted thread's budget. Finally, at thread creation, it should decrease by an amount *at least* equal to the new thread's commitment. The inductive argument then clearly applies to the new sub-transitions. We only note here that finding the right `buvecCommitment` is non-trivial; it requires close inspection of exactly how the "flow" of budget happens. Once this is done, the rest of the proof reduces to a straightforward PVS exercise. The theory for this model now contains 1979 lines of PVS code and 254 lemmas.

4.2 Adding Slacks

Changes to the Model

With the introduction of slack, we need to make a number of changes to the model. First, we need to differentiate between two types of threads depending on whether they are slack-enabled or not. The `state` record type now has a new field: `netSR`. It is an array of time values, also indexed by the system's periods. While the purpose of `buvec` is to keep track of available periodic budget for thread creation, the purpose of `netSR` is to keep track of time *temporarily available* for slack-enabled threads. `netSR` is initialized with zeros, except the element indexed by the fastest period is set to the *timeline slack*.

The definition of transitions are updated as follows: Whenever a thread `completes` or `deletes` itself, any unused time will be added to the element of `netSR` indexed by that thread's period. When a slack-enabled thread of `period` has spent its entire budget, it may request to continue to run on slack time. The request is granted if slack is available (resulting in the sub-transition `switchToSlack`), and denied otherwise (resulting in the sub-transition `timerInterrupted`). The amount of available slack is computed by summing the elements of `netSR` with index at most `period`. Finally, the sub-transition `sopReset` also needs to be updated to reflect the updating of `netSR`.

Changes to the Proof

The changes to the proof are analogous to those carried out when we add dynamic threads to the basic model. We need to add a new term, `slackCommitment` to the inequality defining `goodCommitment`:

```
goodCommitment(s,period):bool=
  commitment(s,period)+
  buvecCommitment(s,period)+
  slackCommitment(s,period)<=remainTime(s,period)
```

Whenever a thread completes or deletes itself, its unused time will be added to `slackCommitment` (as opposed to simply "lost" in previous models). In a sense, it is transferred from `commitment` to `slackCommitment`. The reverse transfer occurs when a slack-enabled thread gets to run on slack. While intuitively clear, finding the right definition of `slackCommitment` also required careful inspection of the scheduler's algorithm.

Notice the incremental update to the definition of the key predicate `goodCommitment`! Each new feature added to the model has a corresponding term that accommodates the changes required to inductively prove TP. The additional amount of work required to prove TP in this full model is comparable to that required when we added dynamic threads to the basic model. We are confident that this model is scalable with even more features such as interrupt threads.

5 Related Work

There are several examples of formal verification applied to real-time schedulers or operating systems in the literature, but as far as we are aware, no verification has been attempted of a real-time system with as much complexity

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

and features as Deos. Fowler and Wellings [8] apply PVS to the formal development of an Ada tasking kernel, which is a substantial example, but makes simplifying assumptions. Unlike Deos, the kernel formalized in [8] does not support dynamic task creation or deletion, or slack scheduling. On the other hand, inter-task communication via Ada protected objects is modeled. A similar mechanism is present in Deos but its addition to our formal model remains the subject of future work.

A PVS formalization of a real-time resource allocation algorithm — the priority ceiling proocol — is presented in [6]. The model used in this work is similar in many ways to our Deos model. The priority ceiling protocol is specified as a state-transition system with a discrete time model, and proofs rely on finding inductive invariants. In earlier work, Wilding [18] used the ACL2 theorem prover to verify one of Liu and Layland's theorems [11], a fundamental result of real-time scheduling theory. Yuhua and Chaochen [19] too have constructed proofs for these schedulability theorems.

As illustrated in this paper, the use of deductive methods supported by an interactive theorem prover, is a powerfull technique for obtaining high confidence in the correctness of a real-time scheduler. More automatic methods, based on model checking, have also been investigated in this context. For example, Vestal [17] models the MetaH executive using hybrid linear automata and uses reachability analysis to verify timing properties (i.e., deadlines are met) and other invariants (e.g., all variables are initialized before they are used). The analysis discovered several defects in the executive and in MetaH tools. The verification examines several scenarios, each consisting of one or two tasks with different timing characteristics and synchronization constraints. Each of these simple scenarios is analyzed using state-exploration algorithms. Using the same type of automata, Altisen et al. [1] propose a general methodology for examining scheduling issues, formulated a supervisory control problems. Campos at al. [3] describe an extension of symbolic model-checking for finite-state real-time systems. Model checking is used to determine the minimum and maximum execution times for all processes, which are modeled as state-graphs. This information is used to verify the schedulability of an aircraft control system with respect to a rate-monotonic scheduler.

6 Conclusions

Complex safety-critical avionics systems are an ideal application for formal verification techniques. Our previous work has shown that model-checking is effective at identifying bugs in such systems, but may be limited by memory requirements. When we started the work reported in this paper, our goal was to employ the power of theorem proving to enhance the validity of the results obtained from model-checking of Deos. Our first step was to build an abstract PVS model of Deos with the same capabilities as our more concrete Spin model. In addition, to ensure the correctness of our model, we had to show that the PVS model provided the same time partitioning guarantees as the Spin model. This is the first time that a scheduler design of this complexity has been formally verified.

Having experienced the difficulty of modeling and proving properties about large systems using theorem proving, we realized at the outset that our approach had to be scalable. Our solution was to use a state-transition system, explicit clock and inductive proofs. Our results with incrementally adding two new features to a base model have shown that this approach is indeed scalable. As each new feature was added, more than 95% of the model remained essentially the same, and 95% of the proofs were reused without modifications (except for the name changes mentioned above). Most of the work in adding features revolved around adding new states corresponding to those features, adding transitions from and to those states, and proofs over those new transitions. The toughest issue was the identification of new time partitioning invariants, which continues to be a difficult art. However, our feature-based approach to proof decomposition and the use of disjunctive invariants makes it possible to develop sufficiently strong inductive invariants in a systematic way. It might be interesting to formalize this approach of feature-based decomposition in terms of rely-guarantee rules.

Our next task is to use the PVS model to prove abstractions in our Spin model. The properties that will be investigated include extension of results obtained in Spin for three threads to arbitrary number of threads supported by PVS. Since we now have a concrete Spin model and an abstract PVS model of the same system, we have the platform to study the interplay between these two techniques and to identify means for maximizing their respective strengths.

References

[1] K. Altisen, G. Gößler, and J. Sifakis. A Methodology for the Construction of Scheduled Systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Pune, India, September 2000. To appear.

[2] P. Binns. A robust high-performance time partitioning algorithm: the digital engine operating system (deos) approach. In *Proceedings of the 20th Digital Avionics System Conference*, 2001.

[3] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing Quantitative Characteristics of Finite-State Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 266–270, 1994.

[4] D. Cofer and M. Rangarajan. Formal modeling and analysis of advanced scheduling features in an avionics rtos. In *2nd International Conference on Embedded Software, EMSOFT*, 2002.

[5] D. Cofer and M. Rangarajan. Formal verification of overhead accounting in an avionics rtos. In *IEEE Real Time Systems Symposium*, 2002.

[6] B. Dutertre. Formal Analysis of the Priority Ceiling Protocol. In *IEEE Real-Time Systems Symposium (RTSS'00)*, pages 151–160, Orlando, FL, November 2000.

[7] B. Dutertre, H. Saïdi, and V. Stavridou. Intrusion-Tolerant Group Management in Enclaves. In *International Conference on Dependable Systems and Networks (DSN'01)*, pages 203–212, Göteborg, Sweden, July 2001.

[8] S. Fowler and A. Wellings. Formal Development of a Real-Time Kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 220–229, December 1997.

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

[10] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992.

[11] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[12] Z. Manna and A. Pnueli. Temporal Verification Diagrams. In *International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765, Sendai, Japan, April 1994. Springer-Verlag.

[13] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *International Conference on Software Engineering*, pages 488–497, 2000.

[14] H. Pfeifer. Formal Verification of the TTP Group Membership Algorithm. In *FORTE/PSTV 2000*, Pisa, Italy, October 2000.

[15] RTCA. *Software Considerations in Airborne Systems and Equipment Certification, DO-178B*, 1992.

[16] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

[17] S. Vestal. Formal Verification of the MetaH Executive Using Linear Hybrid Automata. In *Procedings of the 6th IEEE Real-Time Technology and Application Symposium (RTAS'2000)*, pages 134–144, Washington, DC, May-June 2000.

[18] M. Wilding. A Machine-Checked Proof of the Optimality of a Real-Time Scheduling Policy. In *Computer-Aided Verification – CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 369–378, Vancouver, Canada, June-July 1998. Springer-Verlag.

[19] Z. Yuhua and Z. Chaochen. A Formal Proof of the Deadline Driven Scheduler. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 756–775, Lubeck, Germany, September 1994. Springer Verlag.

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**