USING MODEL CHECKING FOR VERIFICATION OF PARTITIONING PROPERTIES IN INTEGRATED MODULAR AVIONICS

Darren Cofer, Eric Engstrom, and Nicholas Weininger

Honeywell Technology Center, Minneapolis, MN

John Penix and Willem Visser, NASA Ames Research Center, Moffet Field, CA

Abstract

Time partitioning is a crucial property for integrated modular avionics architectures, particularly those in which applications of different criticalities run on the same processor. In a time-partitioned operating system, the scheduler is responsible for ensuring that the actions of one thread cannot affect other threads' guaranteed access to CPU execution time. However, the large number of variables affecting application execution interleavings makes it difficult and costly to verify time partitioning by traditional means.

We believe that automated model checking is a promising technique for verifying the correct design of partitioning algorithms. Our experience with modeling the DEOS scheduler shows that expressive models can be produced at a reasonable cost. Using automated model checking can increase design assurance by allowing coverage of a larger range of execution interleavings than can feasibly be covered by traditional testing. Furthermore, model checking can decrease development and testing costs by finding design errors early in the development cycle.

Introduction

Increasing aviation safety and reducing delay are two of the greatest challenges facing the aviation industry for the next 10 years. These objectives, however, are in tension with each other. Increased system congestion means that more aircraft are departing and arriving at busy airports during peak traffic times. This increases the chance that human error or component breakdown could lead to an incident or an accident. Increased congestion also increases pilot and controller workload, which can reduce the operational safety margin.

There are several initiatives, either recently deployed or in development, that affect both delay and safety. Delay/capacity initiatives include Reduced Vertical Separation Minimum (RVSM – decreasing the vertical separation requirements for trans-oceanic corridors), Airborne Information for Lateral Spacing (AILS – increasing the capacity of airports with closely spaced parallel runways under instrument landing conditions), Collaborative Decision Making (CDM – optimizing utilization of airports during poor weather conditions), and the emerging Free Flight initiative. Safety initiatives include the Enhanced Ground Proximity Warning System (EGPWS – a predictive ground avoidance system), Controller/Pilot Data Link Communication (CPDLC), Cockpit Display of Traffic Information (CDTI) and Airborne Weather Information (AWIN).

These developments will result in the addition of new systems into the on-board avionics of both existing and newly developed aircraft. New radio standards, critical databases, display systems and information management and decision systems will be added to support the required on-board functionality. These avionics changes may be met by the addition of totally new equipment, or more likely, through the modification of existing equipment. Given the simultaneous push for capacity and safety improvements, it is critical to ensure that these system changes do not have unintended consequences.

In addition, over the past decade Integrated Modular Avionics (IMA) has gained popularity as a more cost-effective method (reducing size, weight, power and recurring cost) of fielding advanced avionics systems. IMA systems use a shared resource environment to simultaneously host functions of differing criticality. This makes such platforms a natural location to place new functionality. It also places a special burden on the

0-7803-6395-7/00/$10.00 ©2000 IEEE

IMA operating system to keep functions of different criticality levels from interfering with each other.

Sharing resources, including both processing and I/O, involves a large amount of concurrency. Software threads interact with each other in complex patterns. Correct interaction between these threads is ensured by the protection mechanisms and scheduling functionality of the real time operating system. Under a cooperative research agreement between Honeywell and NASA's Langley Research Center we will develop techniques to verify time and space partitioning properties in IMA operating systems to better ensure the integration of new functionality in a safe manner. We will then extend those techniques to enable the verification of IMA applications. We believe that the use of formal techniques to increase IMA software design assurance will be crucial to aviation safety over the next decade.

The remainder of this paper provides an overview of our work to date in formal verification of IMA partitioning properties and describes our plans to continue this effort.

Initial Work on Time Partitioning

The Digital Engine Operation System (DEOS) was developed by Honeywell for use in our Primus Epic avionics product line. DEOS supports flexible IMA 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.

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 for obtaining FAA certification is to develop and test the software in accordance with the guidelines in RTCA document DO-178B [1] which uses structural coverage as a measure of testing adequacy. These structural coverage requirements are not only expensive to achieve, but they are ineffective in identifying certain classes of errors, especially those involving timing or race conditions.

The DEOS development team 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 design review process included checking manually that the pre-conditions, post-conditions, and invariants were satisfied by the implementation. Several very subtle errors were detected that the developers believed would have been impossible to detect without these techniques. As a result, they became interested in increasing the formality, reliability, and efficiency of the review process by using automation.

Honeywell and NASA Ames began a collaboration to investigate techniques that will enable automated tools to be employed as part of the ongoing software development process. The specific technique investigated was *model checking*, a formal verification technique for finite-state concurrent systems [9], [4], [3]. Unlike testing, model checking 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.

Honeywell engineers and researchers at NASA Ames collaborated to produce a model for use with the Spin model checking tool developed by Holzmann at Bell Labs [9]. The model was translated from a core "slice" of the DEOS scheduler. This model was then checked for violations of a global time partitioning invariant, using Spin's automated state space exploration techniques. We successfully verified the time partitioning invariant over a restricted range of thread types. We also introduced into the model a subtle scheduling error, originally discovered and fixed during the standard DEOS review process; the model checker quickly detected that the error produced a violation of the time partitioning invariant.

DEOS Overview

DEOS is a microkernel-based real-time operating system designed for IMA architectures.

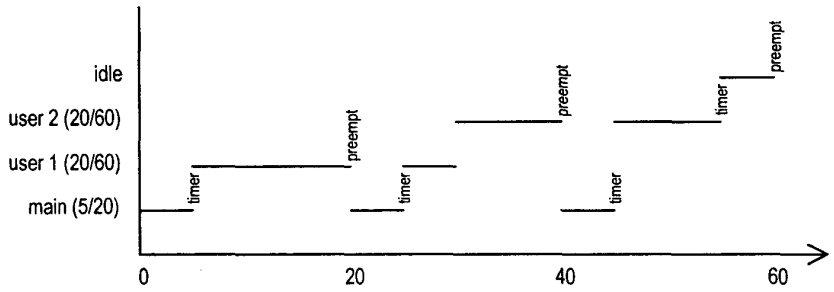

Figure 1. Thread scheduling timeline

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

The DEOS scheduler enforces time partitioning using a Rate Monotonic Analysis (RMA) scheduling policy [10]. Using this policy, threads run periodically at specified periods, and they are given per-period CPU time budgets, which are constrained so that the system cannot be overutilized [2].

Figure 1 shows an example DEOS scheduling timeline. The example shows a main thread, two user threads, and the 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 as the main thread, each with budget of 20 out of 60 time units. All of the threads have been scheduled and allocated their requested budgets within their respective periods. Threads are interrupted when they use all of the budget (timer interrupt) or when a thread of higher priority becomes schedulable (preemption). The idle thread runs during the remaining time not requested by any thread.

DEOS supports a degree of dynamicism and flexibility not typically found in production real-time operating systems. Features that impact space and time partitioning in DEOS include:

- runtime creation and deletion of threads and processes
- thread synchronization and blocking mechanisms such as mutexes and semaphores
- periodic and aperiodic (interrupt service) threads
- inter-process shared memory areas
- process ownership of memory- and port-mapped I/O resources
- partial orderings in the scheduling of threads at a given period

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 particularly difficult task. Similarly, space partitioning is hard to verify because of the large number of ways in which processes can share data with each other, and because processes can own I/O hardware resources.

While manual review has worked well so far, we recognize that as DEOS becomes even more complex in response to changing user needs, reviewing all of the necessary design and implementation requirements will be increasingly costly and time-consuming.

Time Partitioning Property

As a first step toward formal verification of DEOS, researchers from Honeywell and NASA Ames researchers constructed a formal model from a manually extracted "slice" of the DEOS scheduler using the Spin model-checking tool. We have used this model to verify that the time partitioning property of DEOS holds for this "slice," albeit under a restricted set of conditions. We also introduced into the model a very subtle error that had previously been detected and fixed through manual analysis; the model correctly detected the error as a violation of time partitioning. The results of this work are detailed in [11] and [12].

The Spin model of the DEOS scheduler includes the basic thread scheduling algorithm used in DEOS. The model allows for the dynamic creation of new periodic threads and the dynamic deletion of existing threads. When threads are created within a process, they receive their time budget from the main thread for that process. When they are deleted, the budget is returned to the main thread. The model also allows a thread, in each period, to complete early and suspend itself until the next period, or to use all of the time allotted to it and be forcibly suspended by DEOS.

In addition to the scheduler itself, the model contains concurrent processes representing the idle thread, the main thread, *n* user threads, and the hardware environment combining a system tick generator and the timer process. Communication between the processes is achieved using synchronous message passing.

Within this context, we may express a necessary and sufficient condition for the guarantee of time partitioning:

> *At any given time, and for any given future deadline, the total budget the scheduler is obliged to provide to currently running periodic threads before that deadline does not exceed the actual amount of time remaining before that deadline.*

It is reasonably easy to see that if this assertion is violated, then a scenario can exist in which a thread is denied access to its allotted CPU budget. Verifying time partitioning for this subset of the DEOS kernel is then simply a matter of doing an exhaustive verification of the above assertion using the model-checker. In order to make sure that the state space of the model includes all possible sequences of thread execution, we incorporate into the model a nondeterministic choice of periodic thread behavior in each period (creating a thread, deleting a thread, or suspending itself until its next period).

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) formulas.

We used 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. The second approach relied on verification of liveness properties. A liveness property states that some event (or sequence of events) will eventually occur in the system.

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. To check this we placed an assertion into the code where this value is assigned. With dynamic thread creation enabled, Spin quickly found a violation of the assertion. In the error trace, the main thread and the user thread attempt to execute in the period where the user thread is created, resulting in a CPU allocation greater than 100%.

However, this situation cannot occur in the real system 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 an equal or shorter period, will have reached the end of its period by this time and its remaining budget will get reduced by the amount of time allocated to the user thread, thereby avoiding the time partitioning violation.

In the second experiment, we attempted to verify the following liveness property, which is necessary (but not sufficient) for time partitioning

to hold: If the main thread does not have 100% CPU utilization, then the idle thread should run during every longest period. This condition is captured by the LTL property:

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

That is, it is always the case that when the longest period begins, it will not end until the idle thread runs.

When verification was attempted with two user threads and dynamic thread creation and deletion enabled, Spin reported a violation. The error scenario results when one of the user threads deletes itself and its unused budget is immediately returned to the main thread (instead of waiting until the next period). This bug was, in fact, one which had been previously discovered by Honeywell during code inspections (but intentionally not disclosed to the NASA researchers performing the verification). Therefore, it would seem that model checking can provide a systematic and automated method for discovering subtle design errors.

Future Work

Time partitioning

Our current time partitioning model does not incorporate several important time-related features of DEOS. These include:

- The existence of multiple processes, which serve as (among other things) budget pools for dynamically creating and deleting threads. Time partitioning must be verified at a process level as well as a thread level.
- Several types of thread synchronization primitives provided by DEOS, including counting semaphores, events, and mutexes. These allow threads to suspend themselves or be suspended in ways not accounted for by the current model.
- The existence of aperiodically running threads, used to service aperiodic hardware interrupts. These are budgeted like normal periodic threads, but may run in very different ways.

We propose to integrate these features into the model and verify that time partitioning still holds with these features present. The principal challenge here will be keeping the state space size manageable while increasing the complexity of the model by incorporating these new features. As was documented in [11], the current model has already approached the bounds of exhaustive verifiability on currently available computer systems, although subsequent optimizations have reduced the size of the model somewhat. Furthermore, the current model has only been tested on a small range of possible thread budgets and periods.

We anticipate that the model-checking approach will, by itself, be insufficient to produce a truly comprehensive proof of the correctness of time partitioning in DEOS. It is unlikely that we will be able to produce a DEOS model with all the complexity of the operating system itself, and then conduct an exhaustive verification of time partitioning for that model that covers all possible combinations of thread periods, budgets, and behaviors. If we want to get a comprehensive proof, then, we must break the problem down into smaller pieces.

We will identify a set of "subproperties" such that verification of all subproperties in the set will imply the correctness of time partitioning in DEOS. These might include properties such as:

- time partitioning holds for a particular "representative" set of thread budgets and periods;
- time partitioning holds under the assumption that kernel functions take zero time to execute;
- time partitioning holds if no thread can change its priority.

We can then use appropriate tools to dispatch each of the subproperties. Some might be susceptible to verification by various versions of the existing Spin model; others might be best verified by a very different tool, e.g. a proof verified by a proof-checking system such as PVS. We will then need to construct a "meta-proof," a demonstration that *if* all of the subproperties hold, *then* time partitioning holds for the full DEOS system.

Space partitioning

The feasibility of modeling OS space partitioning properties and verifying the key property with a theorem prover has been demonstrated in [5]. There DiVito presents a generic model of space partitioning for a multithreaded operating system and proves its correctness using PVS. This model includes an inter-partition communication mechanism, and considers kernel handling of data as well as application handling. The approach used is one of noninterference from a security standpoint: applications from distinct partitions must not influence each other's operation or their view of data.

We plan to build on the approach in [5] to construct and verify a model of space partitioning in DEOS. Such a model would be built around processes, which are the units of DEOS space partitioning; each process can have multiple threads, which are not space-partitioned from each other. Each process has a certain range of memory, and a certain number of discrete I/O devices, to which it has access. The key property to verify is that one process cannot influence the operation of another process by writing to devices or memory to which it does not have access.

DEOS contains several important features that will require the extension of its space partitioning model well beyond the scope of that presented in [5]. For example, DEOS includes shared memory blocks to which multiple processes may have read and/or write access. Shared memory may, obviously, cause space partitioning to be violated if it is used carelessly; the important and difficult thing is to verify that *if it is used correctly*, shared memory will not violate space partitioning. Also, DEOS processes contain numerous kinds of objects; a process that owns one of these objects may grant access to the object to another process. These objects must be protected against unauthorized access, and in particular, deletion of a process must not be blocked due to persistent attempts at unauthorized access to an object it owns.

We will need, then, to develop a partitioning property which states that the execution trace of a process is equivalent to that which it would be in a federated architecture, *except* for that portion of the process' state to which access has been granted to other processes. This is somewhat similar to the IPC state developed in [5], but is likely to have more extensive impact on the model definition and the resulting proof structure. Once this partitioning property is identified, we can use it to construct a system-wide proof of correctness in the same manner as for time partitioning.

As with time partitioning, it may well be the case that one model or theory alone is not sufficient to verify a system-wide space partitioning property. We anticipate, then, that we will break down the space partitioning property into numerous subproperties that may be dispatched by different kinds of tools. Once again, the best approach is likely to involve integration of theorem proving and model checking techniques.

Implementation verification

While the models we propose to develop in the previous two approaches seek to verify a far-reaching property over all possible executions of the DEOS system, automated verification of pre- and post-conditions involves checking a multitude of relatively small and self-contained properties. This calls for a quite different approach, as we can see by examining the manual method currently used to check these conditions. A reviewer verifying that, for example, a function `foo()` is always called within a critical section, does not need to construct a mental model of the whole DEOS system; the vast majority of the system is irrelevant to the satisfaction of the condition. He or she needs only to look at the places where `foo()` is called, and verify that either

- the caller enters a critical section before calling `foo()`, or
- the calling function itself is always called within a critical section, and that critical section remains in force until the call to `foo()`.

This example illustrates not only that the portion of the operating system relevant to checking one condition is small, but also that determining the extent of that portion is nontrivial. In order to check that a call to `foo()` occurs within a critical section, we must keep track of whether each line of code, from the beginning of the calling function to the invocation of `foo()`, is in a critical section. This

may become a substantial task if the calling function invokes other functions before it invokes `foo()`. It also requires that we identify which code statements have an effect on critical sections, and consider only those, abstracting away from everything else. For more complicated conditions—e.g. that a function is called only when the value of a certain variable is greater than 100—the abstraction process becomes trickier.

Significant work has been done on the problem of "slicing" code so as to extract only those parts relevant to the definition of a given variable or the truth of a given condition. Dwyer et al. [7] have identified a slicing criterion usable for a simple flowchart-based language which is nevertheless expressive enough to encode most structured programming constructs. Their slicing method is based on analyzing the dependency paths leading into and out of a set of nodes of interest $\{n_1...n_k\}$, and using those dependencies to construct a minimal "residual program" which behaves the same as the full program with respect to $\{n_1...n_k\}$. We propose to apply these slicing techniques to develop an automated mechanism for abstracting out the parts of DEOS code which are applicable to any given precondition or post-condition. We then propose to translate the sliced code into the modeling language of a model-checker such as Spin. Once this is done, we can state the desired condition as an assertion, and run the model-checker to verify that the assertion holds. This process will then constitute an automatic verifier for pre- and post-conditions in DEOS.

The translation piece of the process, like the slicing, will build on existing work. One possible solution would be the translation of DEOS code into Java. DEOS uses a well-behaved subset of C++ that may be amenable to straightforward translation into Java. There already exist tools that can translate Java code into Spin models automatically, or even perform model-checking directly on Java code; one example is Java Pathfinder, a tool developed at NASA Ames and described in [8]. If Java translation proves infeasible, there are other methods we can explore, such as extending existing tools to work on C++ code, or translating into another modeling language.

We are aware of the difficulty involved in solving the slicing problem in the general case, and will therefore build up the DEOS slicing and translation effort gradually, starting with easier special cases. We plan to begin by developing automated slicing and translation for preconditions stating that a function must or must not be called within a critical section, such as that given in the example above. These types of preconditions are among the most common in DEOS. Failure to satisfy these preconditions has been the source of several subtle errors discovered during the development and manual review processes, and the ability to automatically check them would be of immediate benefit.

The verification of more complex preconditions, like the "must only be called when x > 100" example, will rest largely on the ability to specialize code in the slicing process—that is, to partially evaluate much of the data in order to split up code into useful cases. Dwyer et al. [6] describe this specialization. The method used is much like that for slicing, but in reverse: take a certain set of "variables of interest," pre-evaluate them, and generate one or more residual programs containing only that code which depends on parameters still unknown, i.e. not in the pre-evaluated set.

Application verification

DEOS application developers, like developers of any safety-critical real-time software, face problems that are in many ways analogous to those facing the developers of DEOS itself. They must verify that their multithreaded applications operate correctly, without deadlocks or other timing violations, over the wide range of thread interleavings possible in the dynamic DEOS system. They must, in particular, verify that the varying use of thread budgets—i.e. the fact that a thread may take a shorter or longer time to complete its tasks in one period than it does in the next—does not affect the timing properties of their application. Furthermore, DEOS does not *guarantee* that applications will enjoy the benefits of space and time partitioning, but only *enables* them to do so; DEOS cannot guarantee that applications will correctly use the primitives it provides for maintaining partitioning. Thus application developers must ensure that they do not inadvertently grant access to objects to unauthorized processes; share memory in a way that

enables "rogue" processes to corrupt their data; or otherwise violate the rules required to take advantage of DEOS's partitioning capabilities. This verification, again, often cannot be achieved by traditional testing alone, since a test cannot cover all of the situations that might conceivably occur in-flight.

To address this problem, we will leverage our previous work to construct a model of DEOS that can be used as an environment to verify application code. This model would include:

- A representation of (at least part of) the DEOS API, including thread creation and deletion, process creation and deletion, and the thread synchronization primitives.
- A nondeterministic scheduler, based on our current DEOS scheduler model but considerably simpler, designed to model all those and only those thread execution interleavings possible within a running system, and also to model thread access to sharable resources and objects.
- A model of a "ghost thread" or "ghost process" representing applications other than the ones of interest for verification. Such a model is crucial because one of the facts that must be taken into account in testing an application is that one or more unrelated processes might be running concurrently with the application in the system, and they might do anything that DEOS allows them to do.

The representation of the API would then form an interface to application code. Slicing and translation techniques would be used to abstract out only those parts of the application code related to the DEOS API calls, and to translate that code into a format which could be "plugged in" to the DEOS model.

In addition, we would have to develop a means of representing in the model upper bounds on the amount of time taken to execute application code. A typical periodic application depends for its correctness on the knowledge, verified by empirical testing, that certain sequences of code will execute within a certain bounded amount of time. Without this knowledge, verifying timing properties within a DEOS model is impossible, because undesirable thread execution interleavings will exist in the model that cannot exist in the real system due to these time bounds.

Impact on Certification

In [13] Rushby reviews the requirements for software aspects of certification contained in DO-178B [1] and describes various ways in which formal methods could be employed to meet its requirements. The formal verification of IMA OS partitioning properties that we have begun is an example of using formal methods to improve quality control by attempting to identify and eliminate faults in the design. In this case, a formal proof will be one of the "other means" allowed in Section 6.2 of [1] to satisfy the verification process objectives in the case of complex behaviors that are not amenable to testing. Use of formal techniques to identify implementation and application faults is an example of using formal methods to improve quality assurance. In this case it is necessary to identify requirements that state what components and functions should and should not do, as well as what assumptions may be made about their environment. The reviews and analyses of Section 6.3.2 (now done manually) will be performed using formal techniques and tools.

We believe that formal methods are a natural candidate for integration into the certification process. Certification documents such as DO-178B emphasize independent verification of software correctness and the use of comprehensive testing and review to check for the satisfaction of both high- and low-level requirements. A multilevel formal verification approach addresses these certification objectives directly. Just as traditional testing is geared to achieve structural coverage of code, so verification of partitioning properties can be designed to achieve coverage of DEOS features. Just as human reviewers are tasked with checking that pre-conditions and post-conditions to functions are satisfied in code, so automated abstraction tools can be used to check those same conditions.

To begin integration of our formal verification work into the certification process we plan to:

- add formal verification tasks to the DEOS software verification plan;

- conduct formal verification runs of our space and time partitioning models as part of the regression testing process;
- include the analysis results in the package of DEOS certification artifacts presented to the FAA;
- make our results generally available within the aviation community to support the use and acceptance of formal methods in software certification.

The intent of this effort is not to supplant any existing tasks required for certification at this time. Rather, we aim to augment our own confidence that DEOS is correct, and to demonstrate the feasibility of using formal methods in developing and verifying safety-critical software. Our inclusion of formal verification results in certification artifacts will provide the FAA and other certification authorities with evidence that commercial vendors are willing and able to use formal methods in the certification process. In the long term, we anticipate that the techniques we develop may provide the foundation for future certification standards and more aggressive use of formal methods.

Since DEOS runs principally on hardware platforms designed for business, regional, and helicopter aircraft, the frequency of certifications will be greater than is typical for the transport class of aircraft. We will have multiple opportunities to demonstrate the use of formal techniques in certification.

References

[1] "Software Considerations in Airborne Systems and Equipment Certification." RTCA document no. DO 178-B, December 1, 1992.

[2] Binns, Pam, "Design Document for Slack Scheduling in DEOS," Honeywell Technology Center Technical Report SST-R98-009, September 1998.

[3] Clarke, E. M., 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*, vol 8, num 2, pp. 244–263, April 1986.

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

[5] DiVito, B. "A Formal Model of Partitioning for Integrated Modular Avionics." NASA Technical Report CR-1998-209703, August 1998.

[6] Dwyer, M., J. Hatcliff, S. Laubach, and N. Muhammad. "Specializing Configurable Systems for Finite-State Verification." KSU CIS TR 98-4. Available from http://www.cis.ksu.edu/santos/bandera/

[7] Dwyer, M., J. Hatcliff, and H. Zheng. "Slicing Software for Model Construction." *Journal of Higher-order and Symbolic Computation,* to appear. Available from http://www.cis.ksu.edu/santos/bandera/

[8] Havelund, K. and T. Pressburger. "Model Checking Java Programs Using Java PathFinder." *International Journal of Software Tools for Technology Transfer,* to appear. Available from http://ase.arc.nasa.gov/havelund/jpf.html

[9] Holzmann, G.. "The model checker Spin." IEEE Transactions on Software Engineering, vol 23, num 5, pp. 279–295, 1997.

[10] Liu, C. L. and J.W.Leyland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment." *Journal of the ACM* 20(1), January 1973, pp. 46-61.

[11] Penix, J., 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.

[12] Penix, J., W. Visser, E. Engstrom, A. Larson, and N. Weininger. "Verification of Time Partitioning in the DEOS Scheduler Kernel." ICSE 2000.

[13] Rushby, J. "Formal methods and their role in the certification of critical systems." Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and issued as part of the

FAA Digital Systems Validation Handbook (the guide for aircraft certification).

[14] Rushby, J. "Partitioning for safety and security: Requirements, mechanisms, and assurance." NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.