Verifying Time Partitioning in the DEOS Scheduling Kernel

John Penix (john.penix@nasa.gov)
Computational Sciences Division, NASA Ames Research Center

Willem Visser (wvisser@email.arc.nasa.gov) and SeungJoon Park
Research Institute for Advanced Computer Science, NASA Ames Research Center

Corina Păsăreanu (pcorina@email.arc.nasa.gov)
Kestrel Technologies, NASA Ames Research Center

Eric Engstrom, Aaron Larson and Nicholas Weininger
Honeywell Technology Center

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 with minimal abstraction could be used to find a subtle implementation error that was originally discovered and fixed during the standard formal review process. The experiment involved translating a core slice of the DEOS scheduling kernel from C++ into Promela, constructing an abstract "test-driver" environment and carefully introducing several abstractions into the system to support verification. Attempted verification of several properties related to time-partitioning led to the rediscovery of the known error in the implementation.

The case study indicated several limitations in existing tools to support model checking of software. The most difficult task in the original DEOS experiment was constructing an adequate environment to close the system for verification. The fidelity of the environment was of crucial importance for achieving meaningful results during model checking. In this paper, we describe the initial environment modeling effort and a follow-on experiment with using semi-automated environment generation methods. Program abstraction techniques were also critical for enabling verification of DEOS. We describe an implementation scheme for predicate abstraction, an approach based on abstract interpretation, which was developed to support DEOS verification.

1. Introduction

The cost of software aspects of flight certification for avionics systems has grown significantly in recent years due to the increased use and complexity of software. This software provides advanced control, communication and safety features at a reduced cost and weight. However, verification and certification of software for high levels of assurance is extremely expensive due to the manual effort needed to support the extensive testing required by the Federal Aviation Administration

(FAA) [37]. Furthermore, the difficulty of verification and certification will continue to increase due to an industry trend toward Integrate Modular Avionics (IMA) to further reduce costs. IMA allows multiple applications of varying criticality levels to execute on a shared computing resource [59]. Part of the cost savings strategy of IMA is that software applications will be individually certified allowing them to be mix-and-matched with avionics platforms. This is currently not supported by the FAA certification process which takes the more conservative approach of certifying each platform configuration. However, this approach is well advised because it is well known that testing is inadequate to assure that arbitrary combinations of applications will operate together safely [12].

Reducing the manual effort required to support certification while increasing the levels of assurance will require significant advances in software verification and certification technology. We have been investigating the use of model checking to support the analysis of critical avionics software systems. Model checking is an algorithmic formal verification technique for finite-state concurrent systems [18, 56]. Originally applied to hardware verification, model checking has become a promising technique for analyzing software requirements specifications [2, 14, 15, 38] and software design models [1, 24, 40]. One reason for this trend is that, at high levels of abstraction, the scalability limitations of model checking can be avoided while providing useful information about a system. This is convenient because early life-cycle errors are expensive to correct later [7] and can often lead to safety critical failures [47]. However, some software errors cannot be discovered in the requirements and design stages. This may be because the details of the system are not sufficiently elaborated to reveal problems until implementation, or simply because errors are made during implementation. NASA has recently suffered from a number of software implementation problems, including a missing critical section that caused a deadlock in the Deep Space 1 Remote Agent control system [35] and a variable that was not re-initialized after a spurious sensor signal that led to the loss of the Mars Polar Lander [45]. These errors are symptoms of the fact that software has become a pervasive component of aerospace systems and is therefore more complex and difficult to design and validate.

The state of the art for finding errors at the implementation level are static analysis [53, 64, 31, 28] and testing [6, 13]. However, testing only provides a small degree of behavioral coverage of a system, especially for concurrent systems, where testing has limited control over thread scheduling [70, 43]. Static analysis has better success dealing with concurrency, but it can be challenging to obtain accurate results [49]. Model checking, however, has the potential to provide more extensive behavioral coverage in two ways. First, the model checker can evaluate every possible interleaving of threads in the system. Second, model checking can use nondeterministic environment models to close a system for verification. This enables the model checker to generate all combinations of environmental behaviors as the closed system is checked. While in practice it is not possible to exhaustively search this space of behaviors, it provides a comprehensive starting point for systematic reduction and abstraction of the state space.

This paper describes the analysis of a time partitioning property of Honeywell's Dynamic Enforcement Operating System (DEOS) scheduling kernel, using the SPIN model checker. The goal of this experiment was to investigate whether model checking, supported by minimal, well-defined abstractions, could be used to find a subtle implementation error that was originally discovered and fixed during the standard formal review process. The analysis was done on a model of the system very similar to the original code: there is essentially a 1-to-1 mapping from statements in the original code to statements in the model. Therefore, this work can be classified as one of the first attempts at *program model checking* (or *software model checking*) [3, 22, 39, 42, 68]. The philosophy of program model checking is that programs written in popular programming languages should be model checked directly in a (semi-)automated fashion. The entire process used in the investigation is shown in Figure 1. To best understand the feasibility and applicability of this approach, the process of translating the source code into a model checking language was separated from the process of abstracting the code to permit tractable verification. This allowed us to assess the type and extent of abstraction that might be required to apply model checking directly to source code.

During this investigation, we addressed several challenges of model checking complex software systems. First, in order to analyze the kernel, it was completed with an *environment* that adequately models user threads running on the kernel, the hardware clock and the system timer. Second, the state space of the kernel is very large (exhausting 4 Gigabytes of memory during verification, without completion), so *abstraction* was required to make verification tractable. The main contribution of this paper is to demonstrate that model checking can be used to locate subtle errors in complex software systems. A second contribution is to motivate and demonstrate how tool support for environment generation and data abstraction can make these techniques more cost-effective so that they may be used in practice.

The most difficult task in the original DEOS experiment (Section 2) was constructing an adequate environment to close the system for ver-

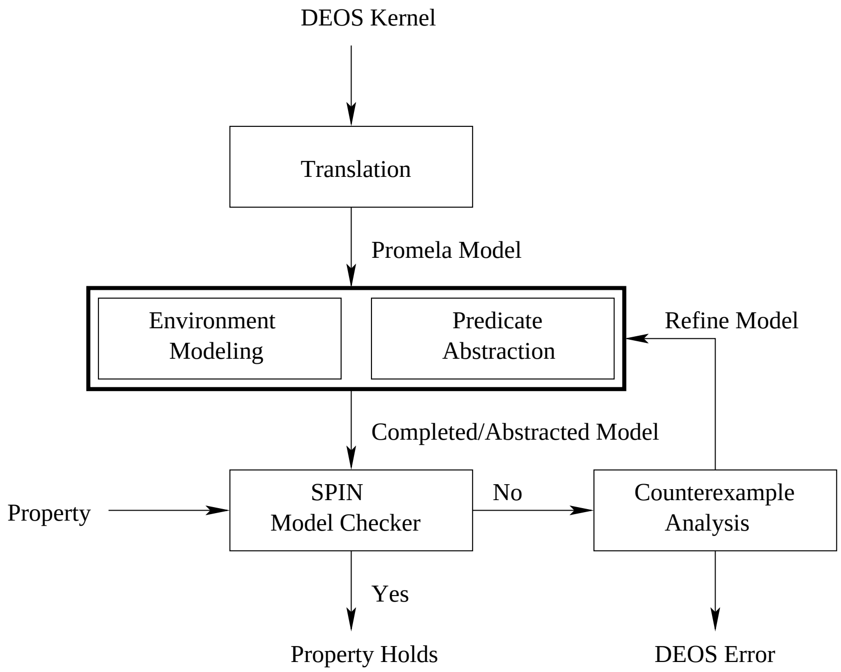

Figure 1. Methodology used to investigate source code model checking.

ification. The fidelity of the environment turned out to be of crucial importance for achieving meaningful results during model checking. 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. In this paper, we describe the initial environment modeling effort (Section 3) and a follow-on experiment with using semi-automated environment generation methods (Section 3.5).

Systematic abstraction also played a critical role in making the verification of DEOS tractable in practice. In this paper, we describe an extension to predicate abstraction, an abstraction approach based on abstract interpretation, to allow it to be used on this case study. Section 4 provides an overview of the use of abstraction to support verification and introduces predicate abstraction. The existing work on predicate abstraction has been in the context of simple modeling and programming languages. The main contribution of our work is the extension of existing abstract frameworks to support abstraction of relationships between classes, or *interclass abstractions*. We show how a specific infinite state programming pattern that occurs frequently in practice can be transformed to a finite state program using predicate abstraction. We then demonstrate how this approach was applied to DEOS to allow tractable verification (Section 4.4).

Since our initial effort to analyze DEOS, a number of subsequent studies, by ourselves and others, have been performed on DEOS with a variety of different approaches. In Section 5 we highlight these activities and also look at related work in the area of program analysis via model checking. Because the problem of extracting models from programs has received much attention [21, 36, 22, 3, 41, 68, 65], we do not present the details of our translation [50] from C++ to Promela, the input language of SPIN. Finally, Section 6 contains conclusions and future research directions.

2. Overview of Original DEOS Experiment

For certification of critical flight software, the FAA requires that functional software testing achieve 100% coverage with a structural coverage measure called Modified Condition/Decision Coverage (MC/DC) [58, 16]. Although MC/DC coverage is quite extensive and expensive to achieve, Honeywell was still concerned that it would not be sufficient to assure complex properties in integrated modular avionics architectures. This concern was based on their experience developing and testing the DEOS operating system. During DEOS development, a subtle error in the time partitioning implementation was not discovered by extensive testing.

To address this concern, we performed an experiment to determine whether model checking, with only minimal abstraction, could provide a systematic method for discovering this error. Honeywell provided an overview of the basic functionality of DEOS and a slice of the operating system containing the budgeting and scheduling algorithms. The NASA team then applied model checking without knowing any details of the DEOS implementation or the error. The source code that was analyzed was 1500 of the approximately 10,000 lines of C++ code which compromise the DEOS kernel. This section introduces DEOS and describes the verification experiment.

2.1. DEOS

DEOS is a portable micro-kernel-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

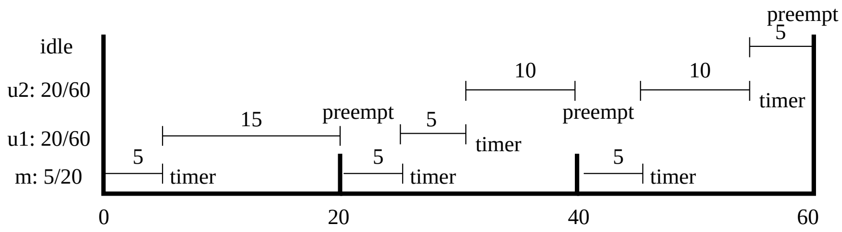

Figure 2. Thread Scheduling in DEOS

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 [59]. 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 [62]. RMA is a general approach for assuring that various system latency requirements can be met during real-time thread scheduling. The basic mechanism in RMA is the assignment of high-priorities to threads with the most stringent real-time requirements. Figure 2 shows an example DEOS scheduling time line. In the example, the system contains a *main thread*, two *user threads* (children of the main thread) 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 as 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. 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.

2.2. Model Checking DEOS

Because there are no model checkers that take C++ as input, the DEOS code had to be translated into the input notation for a model checker. A methodical, 1-to-1 mapping between the code and the model checker input was used to separate abstraction from translation to more clearly understand what abstractions were necessary. We chose the Spin model checker [40] since Promela, the input language for Spin, is the closest model checking language to C++. Promela is a process based imperative language supporting complex data-structures (e.g. records and arrays) and allows communication with shared memory and message passing. An overview of the Promela language, as used in this paper, is provided in Section 3.1. The translation was based on modeling classes as records and using arrays of these records to store object data, similar to the technique used by Havelund and Pressburger for Java [36]. We will not discuss the details of the translation because this has been subsumed by recent work in model extraction [22] and direct model checking [68]. To model check DEOS, an environment was constructed to model the possible behaviors of user threads, the system clock and the system timer. Section 3 describes the environment that was constructed for verification of the kernel.

Verification in Spin involves systematic execution of all possible process interleavings in a program. It detects assertion violations, deadlocks and supports model checking of linear temporal logic (LTL) [51, 67] formulae. In LTL, a pattern of states is defined that characterizes all possible intended behaviors of a system. We describe LTL operators using Spin's ASCII notation. LTL is a propositional logic with the standard connectives `&&`, `||`, `->` and `!`. It includes three temporal operators: `<>`p says p holds at some point in the future, `[]`p says p holds at all points in the future, and the binary p`U`q operator says that p holds at all points up to the point where q holds (p until q).

The main aspect of DEOS that we were interested in verifying was the time partitioning property: that each thread in the kernel is guaranteed to have access to its complete CPU budget during each scheduling period. Two approaches to specifying time partitioning properties in

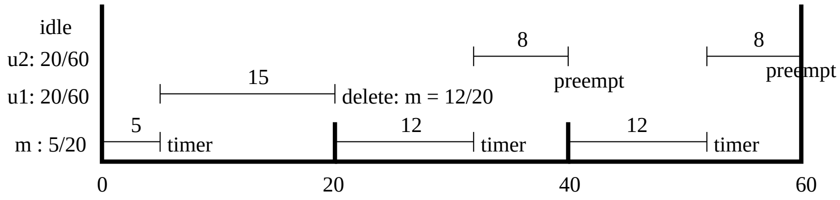

Figure 3. Error scenario

terms of events in the DEOS kernel were investigated. The first, using assertions over program variables, was not effective but led to some insight in the design [50]. The second approach used a liveness property stated in terms of labeled program events referred to from within an LTL property specification. This approach led to the rediscovery of the known error.

To specify time partitioning using liveness, we had to identify a condition that would always occur if time partitioning was maintained. We hypothesized that, in the case where there is slack in the system (i.e. the main and user threads do not request 100% CPU utilization), the idle thread should run during every longest period[1]. 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))
```

meaning that 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.

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 two fold, because not all states are reachable.

The property was checked using several DEOS configurations and environments. In a configuration with 2 user threads and with dynamic thread creation and deletion enabled, Spin reported the error scenario shown in Figure 3. In this configuration, the main thread runs in the fastest period (period 0) with an initial budget of 19/20 time-units. Two user threads are created to run in the next fastest period, period 1, with budgets of 20/60 time-units. To create the CPU budget for each user

[1] This is a necessary, not sufficient, condition of time partitioning.

thread, 7/20 is taken from the main thread, leaving it with a budget of 5/20 time-units. The total budget requested in this configuration is 55/60 time-units, leaving 5 units for the idle thread to fill at the end of period 1.

Figure 3 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 time-units) is given back to the main thread, giving it 12/20 units. 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) time units 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) time-units 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 was the same bug that was discovered by Honeywell during code inspections. This could indicate that model checking can provide a systematic and automated method for discovering errors. However, there were several problems. The state space of the configuration required to show the bug was too large to be exhaustively verified. It was not apparent that the model could even be searched exhaustively to a depth necessary to guarantee discovery of the error. In addition, after adding the fix to the code, we were unable to perform exhaustive verification. To guarantee the error would be discovered and to permit exhaustive verification of the fix, abstraction had to be applied.

3. Environment Modeling

In the original experiment the most challenging task was developing an environment model to allow efficient analysis of time partitioning. The DEOS kernel receives calls from threads that run on the kernel and responds to both periodic system clock interrupts (called system ticks) and timer interrupts from the hardware via interrupt handling routines. The fidelity of the environment was of crucial importance for achieving meaningful results during model checking. Specifically, modeling time in different ways led to trade-offs between result validity and state space size. In this section, we introduce the Promela language of Spin and describe the initial environment modeling effort. We then describe how semi-automated techniques for environment generation, previously only tested on small examples, reduced most of the effort involved in the construction of the original environment model.

3.1. Brief Overview of Promela

A Promela program consists of a collection of processes that communicate via buffered channels and shared global variables. A process body is a sequence of local variable and channel declarations, and statements. Processes can be parameterized with variables, including channels. A process `P` is started with the statement: `run P(...)`. A channel is a bounded first-in-first-out buffer. Processes can read and write messages of a declared type to channels.

Basic statements include assignment statements and channel communication statements. The `skip` statement is a no-operation statement. Statements can either be *executable* or *blocked* in a particular state. Two kinds of statements can block: channel communications (described below) and boolean expressions occurring as statements. A boolean expression blocks if it evaluates to 0 and is otherwise equivalent to `skip`. Statements can be composed sequentially, as in `s1;s2`, and can be grouped together using curly brackets: $\{\ldots\}$. A composed statement is executable if its first statement is executable.

A Promela if-statement has a sequence of options each preceded by a double-colon. Only one of the statements is executed, and only one where the first sub-statement – called the *guard* – is executable. When several statements have executable guards, the choice of the statement is non-deterministic. When no guard is executable, the if–statement blocks. The special `else` statement can be used at most once as the first sub-statement of an option, and it will become executable if all other options are non-executable. There is a corresponding do-statement which is executed repeatedly until a `break` statement is encountered.
Processes communicate over channels using send and receive statements. For example, a process sends the value 5 to a channel `c` by executing the statement `c!5`, while another process can receive this value in the variable `x` by executing `c?x`. If a channel is full, then a send-statement will block. Similarly, if the channel is empty, a read-statement will block. If the size of the channel is defined as 0, communication is by *rendezvous*; the sending process blocks until a receiving process reads the value, and vice versa.

3.2. The DEOS Kernel

Figure 4 illustrates the Promela environment constructed to model check the DEOS kernel. There is a box for each concurrently executing

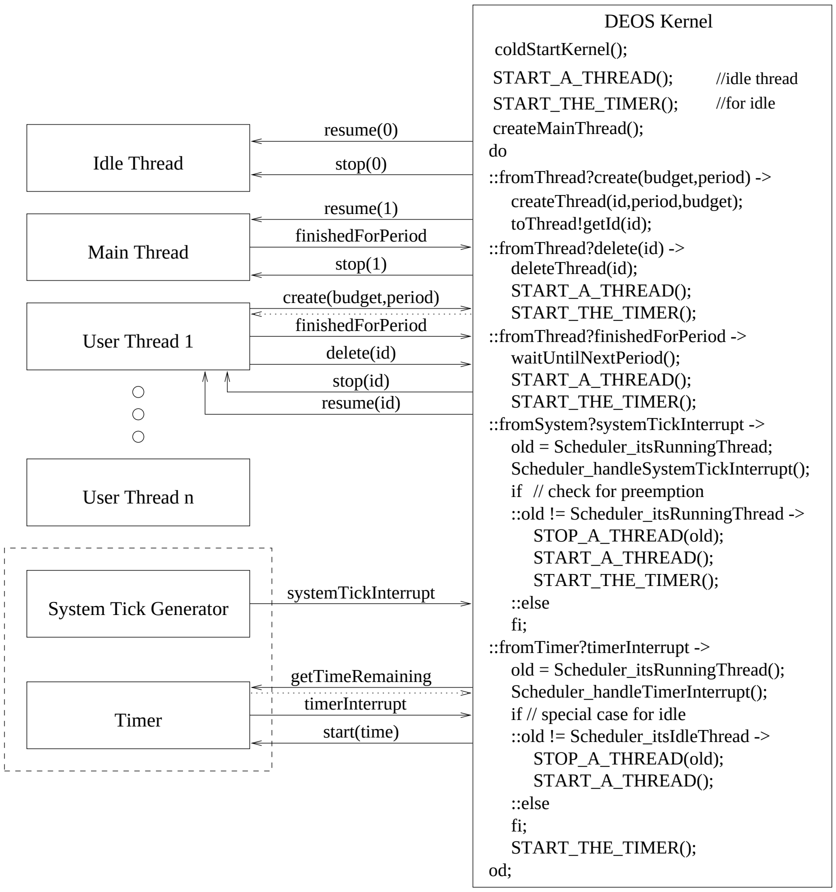

Figure 4. DEOS Kernel and its Environment

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 is to indicate that the system tick generator and the timer were eventually combined into one process. Rendezvous communication between processes is achieved using synchronous message passing, illustrated by the labeled arrows in the figure. Dotted arrows indicate values being returned in response to some messages. In the following sections, we discuss the different components of the DEOS kernel and its environment in detail.

It is important to note that in the real system, there are not separate "processes" for the DEOS kernel and the threads: there is really one thread of control with context switches initiated by kernel code to switch threads. However, this style of scheduling did not map well onto the Promela scheduling semantics. Therefore, the kernel (translated code) and the threads (environment code) were put into different Promela processes and context swaps were modeled by sending start and stop messages to the thread processes.

The kernel code interacts with its environment through a wrapper that maps messages from the environment to methods in the translated code. The wrapper code is shown inside the DEOS kernel box in Figure 4. After initializing the kernel and starting the idle thread and main thread, the process sits in a loop and reacts to messages from the environment.

The kernel can receive three messages from a thread, directly corresponding to DEOS API calls: `create`, `delete` and `finishedForPeriod` (yield). The kernel can also receive interrupt messages which evoke the interrupt handler methods of the kernel. The `systemTickInterrupt` message is generated periodically (at the frequency of the scheduling period of the highest priority threads) and indicates that a thread of higher priority than the currently executing thread may become schedulable. A `timerInterrupt` message indicates that a thread has exhausted its budget and must be stopped immediately.

In response to messages, the kernel can send messages to start and stop threads and to start the timer for a specific amount of time. For example, in DEOS, only the currently running thread can delete itself, so a new thread must be scheduled in response to the `delete` call. If a thread is preempted, it is important to find out how much time still remains from its budget, since it may get another chance to run within the current period. This is done by sending a `getTimeRemaining` message to the timer, and the value returned in the reply is used to update the thread's remaining budget data.

3.3. Threads

We distinguish among three types of threads: the idle thread, the main thread and user threads. The `User` threads have most functionality: they can be stopped, can yield the CPU and can decide to terminate. The Promela code for the user threads is shown in Figure 5. Messages are sent as data records over channels, where the first field in the record denotes the message type. Messages have the form *channel!message_type, data, data*. For message types where some data is unnecessary, `0`'s are used as place holders. In the figure, a nondeterministic `if` statement is used to implement a concise environment model where all possible thread behaviors are examined. Synchronization is used to ensure that when the kernel sends a `resume(id)` message only the thread with the corresponding id will receive it[2]. The simplicity

[2] `eval(id)` allows synchronization only when the message data matches `id`.

of this model can be contrasted to the complexity of the interrupt generator presented in the next section.

```
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 5. The DEOS User Thread Model.

3.4. Interrupts

Modeling the generation of hardware clock and timer interrupts was the most difficult part of constructing the environment for DEOS. Promela and Spin do not provide special support for real-time clocks, so the timers had to be modeled explicitly. The challenge was to determine the level of abstraction at which real-time needed to be modeled.

To verify the time partitioning features of the DEOS kernel, the time-related interrupts had to be coordinated to avoid "impossible" behaviors. Without coordination, system tick interrupts might occur several times, each indicating that 20 time units had passed, but the timer, set for 10 time units, would never go off. To allow the necessary level of coordination, the `SystemTickGenerator` and `Timer` were combined into one process, shown in Figure 6. The combined timer model keeps track of the time that has been used in a period and makes sure that a system tick interrupt occurs only when the appropriate amount of time has been used.

Promela does not directly support sending and receiving messages based on evaluating a condition, but this can be implemented using a two element array of channels, with the condition used as the index

```
proctype TIMER() {
   byte Remaining_time = 0;/* time remaining for thread after timer counted
                              down from Start_time */
   byte Used_time      = 0;/* time used in period since last tick; must be
                              less than or equal to uSecsInFastestPeriod */
   byte Start_time     = 0;/* time the timer was started with */
   byte Y=0;               /* time used by a thread */
   bool tick_since_start = FALSE;
   bool started=FALSE;
   bool timer_went_off = FALSE;

   do
   /* Start Timer */
   ::Sched2Timer?start,Start_time ->
       tick_since_start = FALSE;
       started = TRUE;
       timer_went_off = FALSE;

   /* Get Time Remaining */
   ::Sched2Timer?getTimeRemaining,0 ->
       started = FALSE;
       if
       ::tick_since_start ->
           Timer2Sched[1]!timeRemaining,Remaining_time;
       ::timer_went_off ->
           assert(Remaining_time == 0);
           Timer2Sched[1]!timeRemaining,Remaining_time;
       ::else ->
           /* Y: 0 <= Y <= uSecsInFastestPeriod - Used_time AND */
           /*    0 <= Y <= Start_time                           */
            if
            ::(uSecsInFastestPeriod - Used_time) <= Start_time ->
                Y = uSecsInFastestPeriod - Used_time;
            ::((uSecsInFastestPeriod - Used_time)/2) <= Start_time ->
                Y = (uSecsInFastestPeriod - Used_time)/2;
            ::Y = 0;
            fi;
            Remaining_time = Start_time - Y;
            Timer2Sched[1]!timeRemaining,Remaining_time;
            Used_time = Used_time + Y;
        fi;

    /* Timer Interrupt - channel array trick for conditional send */
    ::Timer2Sched[started]!timerintrpt,0 ->
        Remaining_time = 0;
        Used_time = (Used_time + Start_time)
        timer_went_off = TRUE;

    /* System Tick - channel array trick for conditional send */
    ::Tick2Sched[((Start_time+Used_time)
            >= uSecsInFastestPeriod) && started]!tickintrpt,0 ->
        Y = uSecsInFastestPeriod - Used_time;
        Remaining_time = Start_time - Y;
        Used_time = 0;
        tick_since_start = TRUE;
    od
}
```

Figure 6. Final DEOS timer model

to select which channel will be used for communication.[3] For example, for a condition `p` and channel array `c`, a conditional send is `c[p]!x` and the corresponding receive is `c[1]?y`. These two statements will only synchronize if `p` is true (i.e. equal to 1). This technique was used to control whether a timer interrupt or a system tick message would be sent to the kernel on the `Timer2Sched` channel. A timer interrupt message can only be sent if the timer has been `started`. A tick interrupt message can only be sent when the timer has been started and the amount of time since the previous tick interrupt is greater or equal to the amount of time between ticks (`uSecsInFastestPeriod`).

The behavior of the timer is guided by two variables: the time remaining from the thread's budget (`Remaining_time`) and the amount of time elapsed since the last tick (`Used_time`). These variables are updated in response to each of the messages the timer process can receive as follows:

Start timer - `Start_time` is assigned the value received from the kernel (the thread's budget) with which the timer is started.

Timer interrupt - Indicates that the thread exhausted it's budget, so the `Remaining_time` must be 0. The amount of time used within the period is the previously used time plus the amount of time the timer was started with (`Start_time`).

System tick - The time remaining in a thread's budget (i.e. the time left on the timer when the system tick occurs) must be calculated. First, the amount of time the thread used is calculated, which is the total time in the period (`uSecsInFastestPeriod`) minus the time previously used in the period (`Used_time`). The amount of time remaining on the timer, `Remaining_Time`, is then the amount of time the thread was started with minus the time the thread used. Furthermore, since a system tick just occurred, `Used_time` is reset to zero for the next period.

Get time remaining - 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:

[3] Standard conditional statements cannot be used because the condition evaluation and message command must be executed atomically. Spin does not allow messages to be sent inside atomic sections because messages indicate global states where threads must be interleaved.

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. These cases were selected based on intuition similar to that used in selecting boundary cases during testing, with the middle value included for good measure. Experiments that varied this abstraction showed that the middle value increased the state space by approximately twofold, but did not improve error detection.

3.4.1. *Discussion*

The decision was made to use an abstraction of time, rather than one of the real-time extensions of SPIN (e.g. RT-SPIN [66] and DTSPIN [9]), since we believed the inherent complexity of these techniques would add an unnecessary layer of inefficiency during model checking. Furthermore, the abstractions used were under-approximations (a subset) of timing behavior rather than over-approximations (a superset)[4]. This decision was influenced by the fact that the property being checked (time-partitioning) was dependent on time, and experiments with over-approximation of timing behavior lead to many spurious errors. Verification using an under-approximation of time does not provide a full guarantee that properties are true. However, any errors detected will be real errors, as long as any other data abstractions that are used do not over-approximate behavior. In Section 4 we describe a precise abstraction that can be safely combined with the time under-approximation to preserve errors.

3.5. Environment Modeling Using LTL Assumptions

The most difficult part of defining the environment of DEOS was developing the model for the interrupt generation that would allow us to check time partitioning. Developing and validating the interrupt model in Figure 6 took approximately 2 man-months. Also, despite our best efforts, this model is hard to understand and maintain. Because these issues are serious barriers to the adoption of model checking as a tool to find errors in programs, automated methods for generating environment models were investigated.

This section presents the application of the filter-based methods described by Dwyer et al. [30, 54] to generate the environment that models the generation of interrupts. Starting with the most general definition of the environment, a set of LTL environment assumptions is established and used to refine the environment definition. This refined

[4] See Section 4.1 for more precise definitions of over- and under-approximation.

```
proctype skeleton1_TIMER() {
   byte Start_time;
   byte Remaining_time;
   do
   /* SYSTEM TICK GENERATOR */
   /* send a tick interrupt to the scheduler */
   :: Tick2Sched[1]!tickintrpt,0;

   /* TIMER */
   /* scheduler starts the timer with value Start_time */
   :: Sched2Timer?start,Start_time;

   /* scheduler asks for Remaining_time */
   :: Sched2Timer?getTimeRemaining,0;

   /* timer returns value Remaining_time */
   :: Timer2Sched[1]!timeRemaining,Remaining_time;

   /* send a timer interrupt to scheduler */
   :: Timer2Sched[1]!timerintrpt,0;

   :: skip; /* some internal, non-observable action */
   od
}
```

Figure 7. Timer skeleton

environment is used to rediscover the error in DEOS. Moreover, the environment is precise enough such that when used with the corrected version of the kernel, no spurious errors are reported.

The most general environment for properties stated in LTL is the *universal* environment that is capable of executing any sequence of operations in the system's interface. Under the assume-guarantee reasoning paradigm [52], assumptions about the environment can be expressed in LTL and used to constrain the behavior of the universal environment [54]. In particular, if the environment assumption ϕ and the guarantee ψ are LTL formulas, one can simply check the formula $\phi \rightarrow \psi$. The LTL assumption can also be used to synthesize a refined environment, in which case ϕ is eliminated from the formula to be checked [54].

3.5.1. *Universal Environment for the DEOS Scheduler*

To build the DEOS timer model systematically, the interface between the timer and the scheduler was identified and used to build the environment skeleton in Figure 7. This environment is capable of invoking any sequence of interface operations. However, to verify time parti-

```
proctype skeleton2_TIMER() {
  byte clock = 0;
  byte Start_time;
  byte Remaining_time = 0;
  do
  /* SYSTEM TICK GENERATOR */
  /* send a tick interrupt to the scheduler */
  :: Tick2Sched[1]!tickintrpt,0;
      /* reset */
      clock=uSecsInFastestPeriod;

  /* TIMER */
  /* scheduler starts the timer with value Start_time */
  :: Sched2Timer?start,Start_time;
      /* estimate Remaining_time */
      if
      :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
      :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
      :: Remaining_time = Start_time;
      /* put half_time_option here */
      fi;
  /* scheduler asks for Remaining_time */
  :: Sched2Timer?getTimeRemaining,0;
  /* timer returns value Remaining_time */
  :: Timer2Sched[1]!timeRemaining,Remaining_time;
  /* send a timer interrupt to scheduler */
  :: Timer2Sched[1]!timerintrpt,0;
  od
}
```

Figure 8. Timer skeleton with `Remaining_time` variable

tioning, the model must be refined to capture the relationship between `Start_time` and `Remaining_time`. This was done by introducing a third variable, `clock`, to record the time remaining in a period.

The resulting code for the timer is shown in Figure 8. The underlying principle is *variable time advance* [63, 10]: at each event, the time at which the *next* event will occur is calculated. When the timer is started, depending on the amount of time the timer is started with and the amount of elapsed time in the period, the next event is predicted (either timer or system tick interrupt) and the remaining time and elapsed time is calculated accordingly. Specifically, when the timer is started with a value specified in `Start_time`, the remaining time is estimated as follows:

- If `Start_time` is greater than the current value of `clock`, then a system tick interrupt will occur, so `Remaining_time` for the current

```
proctype U_TIMER() {
  byte clock = 0;
  byte Start_time;
  byte Remaining_time = 0;
  do
  /* SYSTEM TICK GENERATOR */
  /* send a tick interrupt to the scheduler */
  :: Tick2Sched[clock==0]!tickintrpt,0;
      /* reset */
      clock=uSecsInFastestPeriod;

  /* TIMER */
  /* scheduler starts the timer with value Start_time */
  :: Sched2Timer?start,Start_time -> start:skip;
      /* estimate Remaining_time */
      if
      :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
      :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
      :: Remaining_time = Start_time;
      /* put half_time_option here */
      fi;
  /* scheduler asks for Remaining_time */
  :: Sched2Timer?getTimeRemaining,0;
  /* timer returns value Remaining_time */
  :: Timer2Sched[Remaining_time>0]!timeRemaining,Remaining_time;
      timeRemainingGT0:skip;
  :: Timer2Sched[Remaining_time==0]!timeRemaining,0;
  /* send a timer interrupt to scheduler */
  :: Timer2Sched[1]!timerintrpt,0 -> timerinterrupt:skip;
  od
}
```

Figure 9. Timer with restricted rendezvous on `tickintrpt`

thread will be greater than zero, and `clock` is reset to the period duration, `uSecsInFastestPeriod`, to indicate that it will be the beginning of the next scheduling period.

– If `Start_time` is less than or equal to the current value of `clock`, then a timer interrupt will occur, so the `Remaining_time` for the current thread will be zero and the `clock` will be decreased.

– Nondeterministically, the environment can set `Remaining_time` to be `Start_time` and leave `clock` unchanged, which corresponds to the situation that the thread consumed no time.

3.5.2. *Environment Assumptions*

The timer in Figure 8 is still too approximate, leading to spurious counterexamples, because period ticks and resets of the clock can occur arbitrarily. The problem is that `clock` should not be reset unless its value is zero (as would happen in a real system clock). This environment assumption (ϕ_1) can be encoded in LTL as follows:

```
[](tickinterrupt -> !new_tickinterrupt U (clock==0 || []!new_tickinterrupt))
```

Assumption ϕ_1 says that after a system tick interrupt occurs, a new tick interrupt can not occur unless the value of the `clock` is zero. This assumption can more effectively be encoded in the timer model, by restricting the rendezvous on `tickintrpt` to occur only when `clock` is zero, as shown in the code for `U_TIMER` in Figure 9.

Checking the timing property using this restricted environment gives another infeasible counterexample: after a timer interrupt occurs and the kernel asks for the remaining time, the value returned is greater than zero. This would not happen in a "realistic" environment, since a timer interrupt signifies that there is no remaining time left for the current thread.

This situation can be captured by the following environment assumption (ϕ_2):

```
[](timerinterrupt -> !timeRemainingGT0 U (start || []!timeRemainingGT0))
```

This assumption states that, after a timer interrupt occurs and the kernel asks for the remaining time, the environment cannot return a value greater than zero, unless the timer is started again. Notice that in Figure 9, labels were inserted (e.g., `start:`) to define the predicates and we split the rendezvous based on the returned value of `Remaining_time`, e.g. predicate `timeRemainingGT0` is true when the timer thread is at label `timeRemainingGT0`).

When using `U_TIMER` and assumption ϕ_2 as a filter, i.e. when checking the combined formula:

```
([](timerinterrupt -> !timeRemainingGT0 U (start || []!timeRemainingGT0))) ->
   [](beginperiod -> !endperiod U idle)
```

the same error found in the original DEOS experiment is reported. The assumption effectively "filtered out" traces that did not correspond to real executions of the system.

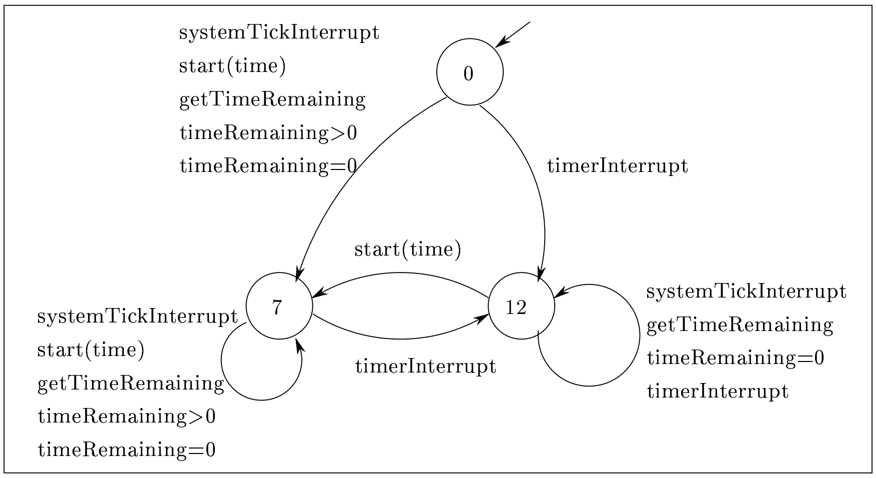

Figure 10. Synthesized assumption graph

3.5.3. *Results*

The time partitioning property was checked using both the `U_TIMER` in Figure 9, with the LTL assumption ϕ_2 encoded in the formula being checked, and, alternatively, with the environment automatically synthesized from ϕ_2.

The synthesis procedure (described in detail in [54]) uses a tableau-like method similar to the one used in SPIN for generating never claims to check LTL properties. The method takes an LTL formula representing the environment assumption, and constructs an automaton that can be represented as a graph (and automatically translated to Ada, Java, or Promela). The graph is a maximal model of the environment assumption, meaning that for every computation which satisfies the assumption, there is a corresponding path in the graph, and that every finite path in the graph is the prefix of some computation that satisfies the assumption.

The state graph for the synthesized environment from assumption ϕ_2 is shown in Figure 10. The labels on the edges denote the allowed interface operations such as rendezvous between environment and kernel, together with the code to be executed for each rendezvous. For example, the label *systemTickInterrupt* is a place-holder for:

```
Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod;
```

Node 0 designates the initial state. The corresponding Promela code is a straight forward implementation of this state machine with additions to update `clock` and `Remaining_time` as in `U_TIMER`.

Verification was done using Spin version 3.2.5a on a SUN ULTRA 60 (360 MHz) with 1G of RAM. following table gives data for each of the model checking runs (using `U_TIMER`, `S_TIMER` and the original `TIMER`). The table shows the total of user and system time in seconds to convert LTL formulas to the Spin "never claim" format (t_{never}), the time to execute the model checker (t_{MC}), the memory used in verification in Mbytes (`mem`) and the number of steps in the shortest error trace.

Environment	t_{never}	t_{MC}	*mem*	*error trace depth*
`U_TIMER`	1:49.97	1.3	3.633	1988
`S_TIMER`	0.1	0.1	2.609	1554
`TIMER`	0.1	0.1	2.609	1619

In conformance with the data from [54], synthesized environments enable faster model checking and better use of memory. The time for generating the never claim with the assumption encoded into the formula to be checked is the dominant time. The time for environment synthesis is negligible, especially considering synthesized environments can be reused across verification runs.

We repeated the experiment with a new version of the DEOS kernel, with the error corrected by the developers (and with the abstraction described in Section 4). Spin exhaustively searched the state space and reported the following data:

Environment	t_{MC}	*mem*
`U_TIMER`	1:38.1	102.289
`S_TIMER`	8.8	23.172
`TIMER`	2.9	12.996

The environments were precise enough so that no errors, real or spurious, were reported. The original, hand-coded environment and the synthesized environment exhibit relatively similar uses of time and memory, compare against the filtered property.

3.5.4. *Conclusions*

This section shows that filter-based environment generation is viable in practice. The effort involved was relatively small, taking one person-week to accomplish, compared to two person-months for the original environment. The environment was built without looking at the code for the kernel; only the code for the original environment was inspected. This second experiment did have the advantage of looking at code that was previously analyzed. The error was known, and also the configuration of the system (i.e. at least two user threads are necessary in order to find the error). However, the environment assumptions were not generated from this previous experience; they were derived systematically from the spurious counterexamples found by Spin.

The most striking advantage of the filter-based approach is the structured way in which environment assumptions are encoded. During the original environment development, assumptions were discovered in much the same way as in the filter-based approach, but these assumptions were just hard-coded into the environment model in an ad-hoc fashion. With the filter based approach the assumptions were first introduced as LTL (filter) properties, and only if the implementation in the actual code was straight-forward, were they added to the code. An environment was also synthesized directly from the LTL filter properties. The synthesized environment performed very similar to the hand-build environment.

In the following section we address the other major barrier to the adoption of model checking in program analysis: the need for automated, or semi-automated, abstraction techniques to reduce the size of the state space that must be analyzed.

4. Program Abstraction for Verification

In the original experiment, the error was detected without introducing abstractions within the DEOS code itself (the abstraction was in the environment). However, it was not possible to guarantee of finding the error or to verify the corrected code. Therefore, abstractions for some parts of the DEOS code were investigated to permit more extensive verification. This section provides an overview of abstraction for program verification, describes predicate abstraction and presents extensions to predicate abstraction to support object-oriented programs. The application of predicate abstraction to DEOS to enable exhaustive verification is then described.

4.1. Program Abstraction

Abstractions are used to reduce the size of a program's state-space in an attempt to overcome the memory limitations of model checking algorithms. Abstractions can be characterized in terms of their effect on a property (or class of properties) being verified, or the way that they approximate the behavior of the system being verified.

An abstraction is *weakly preserving* if a set of properties that are true in the abstract system has corresponding properties in the concrete system that are also true. An abstraction is *strongly preserving* if a set of properties with truth-values either true or false in the abstract system has corresponding properties in the concrete system with the same truth-values. An abstraction is often designed to preserve a single specific property, making strong preservation useful in practice. Nevertheless, abstractions that are only weakly preserving can be much more aggressive in reducing the state-space and therefore are more popular for verification purposes. In practice, the role of verification is often to support rapid and effective debugging during development and evolution. Therefore, we define an abstraction as *error preserving* if a set of properties that are false in the abstract system has corresponding properties in the concrete system that are also false.

A second way to classify abstractions is with respect to the relationship between the behavior of the abstract system and the concrete system. For a reactive software system, program behavior can be defined as the set of possible program execution paths[5], where an execution path is an infinite sequence of program states. *Over-approximation* occurs when more behaviors are present in the abstract system than were in the original "concrete" system. The drawback of over-approximation is that it may add behaviors that invalidate a property in the abstract system that is true in the concrete system. *Under-approximation* occurs when behaviors are removed when going from the concrete to the abstract system. Program testing can be viewed as analysis of an under-approximation: a set of test cases (or a reactive test driver) leads the system through a subset of the possible program executions.

To combine abstraction with model checking, either an abstract state graph is generated during model checking by executing the concrete transitions over abstracted data, or the concrete transitions are abstracted statically (i.e. before model checking) and the resulting abstract transition system is checked. There has been work in automating both approaches by using decision procedures, either during state generation [25, 60] or statically [20, 61]. In the static approach, where abstract transitions are generated, the number of calls to the decision procedures is bound by the size (lines of code) of the concrete system. Because the dynamic approach uses the decision procedures as the state space is explored, it will in most cases require many more calls to the decision procedures than the static approach. However, the dynamic approach can be more precise since it uses information about the current abstract state to determine the abstract transition. This information can be used to eliminate potential next states that cannot be eliminated statically, thus providing a more precise over-approximation. In the

[5] Also called traces or computations.

DEOS study, the static approach was used because it appeared more likely to scale to large programs. The precision problem is addressed by generalizing the statically generated transitions to be defined in terms of multiple abstract predicates, increasing the amount of information about the abstract state that can be used to define the transitions [61].

4.2. Predicate Abstraction

Predicate abstraction, introduced by Graf and Saidi [33], is a form of over-approximation which forms the basis of a number of automated abstraction tools [25, 60, 61, 3, 39]. The basic idea of predicate abstraction is to replace a concrete variable with a boolean variable that evaluates to a given boolean formula (a predicate) over the original variable. This concept is easily extended to handle multiple predicates and, more interestingly, predicates over multiple variables. For example, consider a program with two integer variables, x and y, which can grow infinitely. Since this program will have an infinite state-space, model checking cannot be complete in general. However, closer inspection may reveal that the only relationship of interest between the two variables is whether or not they are equal. We can then define a predicate to represent this relationship, $B \equiv x = y$, and use it to construct an abstract transition system as follows:

- wherever the condition $x = y$ appears in the program we replace it with the condition $B = true$ and

- whenever there is an operation involving x or y we replace it with an operation changing the value of B appropriately.

When generating the abstract transition system, over-approximation can occur when not enough information is available to calculate a deterministic next action or state in the abstract system. For example, the operation `x := x + 1` leads to over-approximation in the abstract transition system (by introducing nondeterminism) in the case where B is false ($x \neq y$) because the concrete result depends upon information that is not available in the abstract state (specifically, whether $y = x - 1$). The abstract transition system for this example showing the effects of the concrete operations `y := x` and `x := x + 1` is shown in Figure 11.

System invariants can be used to construct more precise abstractions (i.e. less nondeterminism and over-approximation). For example, consider the example program in Figure 12, which is an extremely simplified version of part of DEOS. In this program, `x` is first incremented

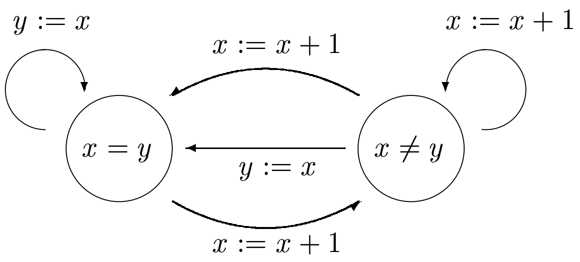

Figure 11. Abstract transition system for $B \equiv x = y$

```
int x,y,z = 0;

  while(true) {
    while(+)
      x := x + 1;
    if (y != x) {
      y := x;
      z := z + 1;
    }
    z := z - 1;
  }
```

Figure 12. Simple example program

one or more times[6] and then `y` is updated with the value of `x`. This program has the property that `z` will always be greater than or equal to zero, because the conditional will always execute. This program will be difficult to model check because `x` and `y` increment indefinitely.

Figure 13a shows an abstract version of the program based on the predicate $B \equiv x = y$, with operations as shown in Figure 11. The result of executing the abstract code corresponding to multiple increments of `x` (the `while(+)` loop) is that `B` will be either `true` or `false`. After exiting this loop, if `B` is `true`, then the conditional will be skipped, and `z` can be decremented below zero. Therefore, this abstract program does not preserve the property $z \geq 0$. However, we can use the program invariant $x \geq y$ to refine the abstraction. This invariant can be used to eliminate the case where $x = y - 1$ and allow the concrete transition `x := x + 1` to always be abstracted to `B := false`. That is, because

[6] The shorthand `while(+)` indicates the loop will non-deterministically execute one or more times.

```
bool B = true;
int z = 0;

while(true) {
  while(+) {
    if (B = true) then B := false
    else B := true or false;
  }
  if (B = false) {
    B := true;
    z := z + 1;
  }
  z := z - 1;
}
```

(a)

```
bool B = true;
int z = 0;

while(true) {
  while(+) {
    B := false;
  }
  if (B = false) {
    B := true;
    z := z + 1;
  }
  z := z - 1;
}
```

(b)

Figure 13. Example program abstracted using criterion $x = y$ (a) alone and (b) with invariant $x \leq y$

`x` is always greater than or equal to `y`, `x` being incremented can never result in `x` and `y` being equal. The refined program, shown in Figure 13b, maintains the invariant on `z`.

This example illustrates a special case in predicate abstraction: when the predicate abstraction does not introduce nondeterminism, over-approximation does not occur and strong preservation is achieved [60]. In the next section we show that a specific class of infinite state programs, occurring frequently in practice, can be transformed to finite state programs by a predicate abstraction that does not introduce nondeterminism.

4.3. Abstractions for Object-Oriented Programs

Most work on abstraction has been on simple modeling or programming languages as both the source and target languages for abstraction. There has been recent work to apply these techniques to `C` programs [42, 3]. However, to apply these techniques to object-oriented systems the following issues must be addressed:

- what kinds of abstractions will be appropriate or necessary for object-oriented programs

- what additional complexity is introduced into the generated abstract transition system (or program) to support these abstractions.

The types of entities that can be abstracted in object-oriented software are variables, classes and relationships between classes. Relation-

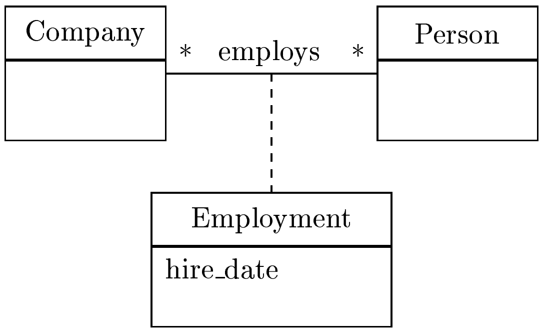

Figure 14. Using Association Classes to Model Association Attributes

ships between classes include relationships between variables or control labels in different classes, and the multiplicity and variability of associations between classes. The behavior of variables and classes can be both under-approximated and over-approximated. Over-approximation of classes can be achieved by extending abstract interpretation techniques for individual variables to object-oriented languages, as implemented in the Bandera toolkit [22]. In Bandera, these techniques have been extended to classes by component-wise abstraction of each field in a class [29]. Under-approximation techniques for verifying object-oriented models are primarily concerned with limiting the number of objects instantiated for a class [44].

These existing abstraction techniques do not support abstraction of relationship between classes. We have developed a technique for extending predicate abstraction to include predicates relating variables from different classes [69]. A problem encountered during that work was determining how to maintain the object-oriented structure of the program when adding abstract predicate variables. Here, we extend our previous approach to support abstraction of more general structural relationships between classes, or *interclass abstractions*, using associations.

In object-oriented modeling and design, *associations* are used to represent structural relationships between classes [8]. For example, the fact that a Person works for a Company can be modeled by an association called "employs" between the class Person and the class Company. If we wish to include a hire date in the model, and there is a many-to-many relationship between companies and people it is not clear where to put this attribute. To solve this problem an *association class* can be affiliated with the association, as shown in Figure 14 [8].

To illustrate how program abstractions can be structured around associations and association classes, we use a simplified example from DEOS. The target of our abstraction is a general pattern where a counter is used to indicate that an event has occurred. This is a form of *abstract time-stamping* that is common in concurrent programming. This pattern consists of an `Event` class containing a counter and any number of `Listener` classes which monitor the occurrence of events by keeping a local copy of the event counter and comparing the two values to determine if the event has occurred. A class diagram representing this pattern is shown in Figure 15a. The `Event` class contains code that increments the counter (T_0 : `count++`), and the `Listener` class contains statements for comparing the local and the `Event` counter (T_1 : `count == last_count`) as well as setting the local counter to the `Event` counter (T_2 : `last_count := count`).

During model checking, this pattern leads to a large state space because the counters increment indefinitely. However, the behavior of the Listener (and hence the system) is only determined by whether these two variables are equal, not by exact values. Therefore, a variable, `p`, defined by the abstract predicate $P \equiv$ `count == last_count`, is introduced. To represent this abstraction, an association class is introduced that encapsulates the new variable, as shown in Figure 15b.

The abstract statements that modify the abstract variable become: T_0 : `p := false`, T_1 : `p` and T_2 : `p := true`. This is a precise abstraction, since similar to the example in the previous section, there is an invariant $count \geq last\_count$, that removes the possible nondeterminism when `count++` is abstracted. In practice, `count` rolls over (say at MAXINT) and this may seem to invalidate our abstraction. However, the correct behavior of the real system implementation also requires the assumption that `count` does not roll over and catch up with `last_count`. Therefore, this abstraction does not introduce any stronger assumptions on the system than those imposed by the implementation, and is therefore a strongly preserving abstraction of the code.

Association classes can be realized by replacing the association class with a standard class that is associated with the original two classes, and removing the original association, as shown in Figure 15c [11]. This approach leads to the creation of a new object to represent each abstracted association. In the case of a single predicate abstraction the result would be an object that encapsulates only one bit. To eliminate this overhead, a slightly different approach to realizing association classes for predicates was used: the new class encapsulates a two di-

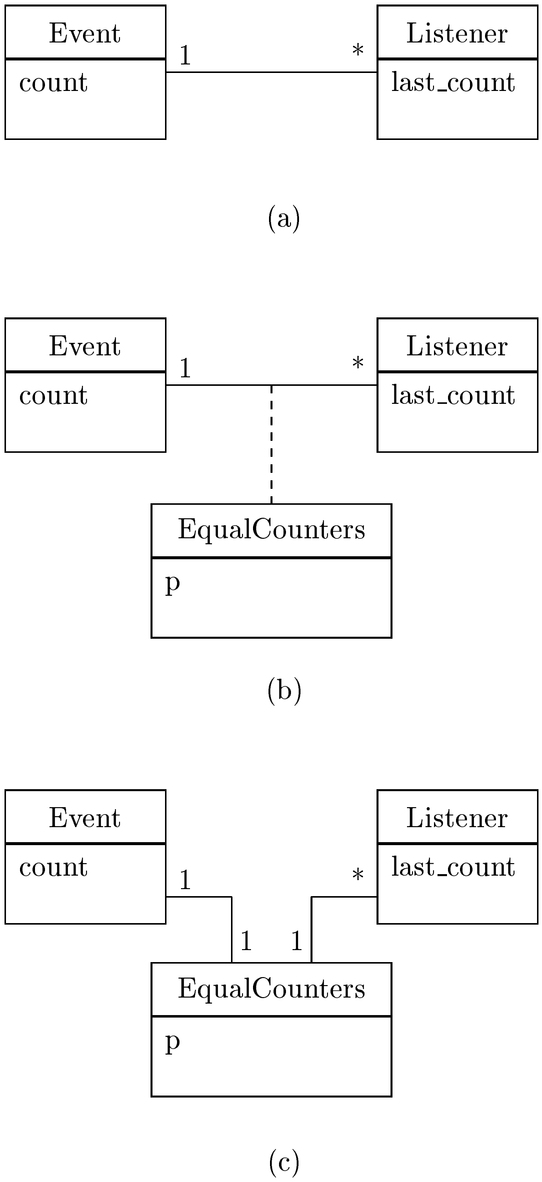

Figure 15. Using Associations to Represent Interclass Abstractions

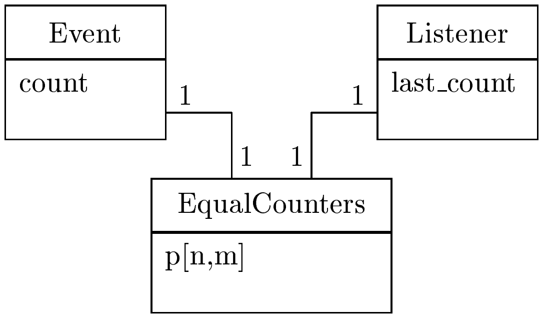

Figure 16. Realizing Interclass Abstractions for Single Predicates

mensional array of bits representing the state of the entire abstract association, as shown in Figure 16, where n and m are the number of Events and Listeners in the system [69] (allowing the more general case of multiple events per listener).

4.4. Abstracting DEOS

In order to allow exhaustive verification of the DEOS model, it was necessary to apply abstraction to the program. The goal was to introduce abstraction carefully and selectively to study how abstraction is applied during verification. First, a simple, ad-hoc abstraction was introduced into the system. Then, this abstraction was refined using predicate abstraction.

4.4.1. *Ad-hoc Abstraction*

The initial abstraction was guided by our intuition 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. However, simulations showed extremely long traces indicating that some data was being carried over these longest period boundaries. Specifically, the `itsPeriodId` data member for the `StartOfPeriodEvent` class is incremented every time the end of the corresponding period was reached. In addition, `itsLastExecution`, in the Thread class, also increases monotonically as it is periodically assigned the value of the `itsPeriodId` counter for the `StartOfPeriodEvent` corresponding to the thread's scheduling period.

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

void Thread::startChargingCPUTime() {
  // Cache current period for multiple uses here.
  periodIdentification 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 17. Slice for `itsPeriodId`

The sections of the DEOS kernel code which involve `itsPeriodId` and `itsLastExecution` is shown in Figure 17. 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. When a thread starts running, `itsLastExecution` is assigned the value of `itsPeriodId` (the return value of `currentPeriod()`, stored in the temporary variable `cp`) whenever the two are not equal. Therefore, `itsLastExecution` will always increase by *exactly* one if a thread is scheduled every period. In this case, both variable types can be replaced with much smaller ranges (namely bits) and still maintain the exact behavior of the system.[7] The DEOS developers assured us that the version of DEOS being considered did, in fact, ensure that a thread is scheduled every period. Therefore, we changed `itsPeriodId` to be incremented modulo 2. This change allowed exhaustive analysis of the entire state space of both the defective and corrected version of DEOS.

[7] The two variables are either equal or different by one, hence only one bit is required to represent the range of possible relationship between the variables.

4.4.2. *Predicate Abstraction*
In the full DEOS system there are synchronization mechanisms, such as events and semaphores, that may cause threads to wait for arbitrary amounts of time (which is not possible in our slice of the system). In this case, our assumption that a thread will execute every period, and consequently the preservation property of the abstraction, breaks down. Therefore, a more general solution was required if the abstraction was to be used in a broader verification context.

It was (eventually) recognized that this part of DEOS is an instantiation of the *abstract time-stamping* technique, introduced in Section 4.3, where `StartOfPeriodEvent` is the `Event` and the `Thread` classes are the `Listeners`. The `Thread` classes check whether a new period has been entered by comparing the `itsPeriodId` of the current period (analogous to `count`) with their own `itsLastExecution` field (analogous to `last_count`). The association based predicate abstraction from Section 4.3 was then used to do a precise abstraction of the DEOS kernel code and combined with the under-approximating timer environment.

It is interesting to note that the abstracted algorithm could not be used as the actual DEOS implementation because, to enable predictive scheduling, execution time spent inside the kernel must be bounded. Updating the array of event bits is proportational to the number of threads in the system, which is not bounded.

5. Related Work

5.1. Continued Work on DEOS

After the original experiment, the model was expanded to analyze slack-scheduling, which allows threads to request "slack" time not used by other threads. With no prior knowledge of Promela or Spin, a Honeywell developer was able to translate and insert the slack-scheduling code into the Promela model within one day. On the first model checking run after making these changes, Spin discovered an error in the new code. The developer had translated a slightly outdated version of the DEOS code, and the error uncovered by Spin had been discovered by Honeywell the previous week. The developer reported that it originally took 3 days to discover what was wrong, whereas with the model checker it was easy to replay and understand the error trace. The original DEOS model has been expanded to be of high fidelity with the current DEOS implementation and is updated with major code updates.

The initial work on DEOS clearly indicated that hand translation was not a practical approach. This motivated the development of Java

PathFinder (JPF) [68], a model checker that analyzes Java programs directly. Subsequently, we translated DEOS from C++ to Java - an afternoon's work - and have used this version to evaluate JPF and Bandera. Using Bandera, backwards dependency analysis from the time partitioning assertion identified `itsPeriodId` (Section 4.4) as a candidate field for abstraction. A *range* abstraction, where values 0 and 1 are concrete and all negative numbers and all numbers greater than 1 map to abstract values, was used to abstract this field. Type inference then determined that two other fields (`itsLastExecution` and `cp` from Figure 17) also required being abstracted to the *range* type. Although this abstraction introduced many spurious errors, JPF was directed to only search for "real" counterexamples to force the abstraction to be precise, and the time-partitioning error was found within a few seconds [55, 29, 34].

5.2. Related Research

Since 1997, there has been an increasing amount of research applying model checking to analyze programs written in popular programming languages. Previously, the program analysis was done by manual model construction before model checking.

The first system to automatically analyze programs with a model checker was Verisoft [32]. Verisoft addresses the state space problem by simply not storing states. It therefore relies on scheduler control and guided search to achieve benefits over testing. Verisoft has been successfully applied in analyzing a number of large systems (see Verisoft paper in this Special issue). A similar state-less model checker that analyzes Java programs was later developed by Stoller [65]. We partially address the state-space problem in DEOS by using under-approximation in the timer model. In this case we are operating similar to stateless model checking, where there is an implicit assumption that it is not necessary to cover all states of the system to have a marked improvement over testing.

Several program model checkers are based on automated model-extraction, where the program is translated into the input notation of an existing model checker. Bandera [22] translates Java programs to a number of back-end model checkers, including Spin [40], dSpin [27], SMV [48], Bogor [57], and JPF[8]. Bandera also supports abstraction by transforming the Java programs to *"abstract"* Java programs which are then translated. JCAT [26] translates Java to Spin and dSpin. FeaVer [42] translates C code to Spin. SLAM [3] translates C code to

[8] JPF works on bytecode classfiles, hence translation here means compile it with a Java compiler

boolean programs as input for the Bebop model checker [5]. FeaVer and SLAM incorporate abstraction methods into the translation process. FeaVer's abstraction is semi-automated, while SLAM uses predicate abstraction [3] and abstraction refinement [4] to automated abstraction during model checking.

To support experimentation with abstractions for object-oriented programs, a prototype tool was developed to automatically generate abstracted programs written in Java. Given a Java program and an abstraction criteria, the tool generates an abstract Java program in terms of new abstract variables and remaining concrete variables. The resulting Java program, implementing the abstract transition system, can be tested or analyzed using a Java model checker. The tool is a prototype and is not advanced enough to abstract a Java version of DEOS. However, it was used to abstract part of a Java version of the Remote Agent software, allowing successful model checking [69].

6. Conclusions and Future Work

In this experiment, the Spin model checker was used successfully to rediscover a subtle error in the time partitioning of the DEOS scheduling kernel that was not uncovered during extensive testing. The initial goal of the study was to show that model checking can augment structural coverage based testing, such as the 100% MC/DC coverage required by the FAA certification process for avionics software. The experiment showed that model checking, augmented by minimal abstraction, could find errors in real programs that MC/DC testing did not. Additional contributions of this paper were to show that filter-based environment generation and predicate abstraction for object-oriented programs can be used effectively to reduce the effort of applying model checking to real programs.

We continue to work on extending the applicability of predicate abstraction and integrating it with related abstraction techniques [22, 29]. We have also recently augmented the Java PathFinder model checker with the capability to do analysis by symbolic execution [46]. This allows the model checker to analyze programs with symbolic data, i.e. where variables do not have concrete values, by using constraint solving to eliminate infeasible paths. This generalizes many abstraction approaches, but comes with several research issues, such as efficient application of widening [23].

In the area of environment generation, the process must be further structured and automated to reduce the cost of applying model checking. Although our results indicate that a filter-based approach is beneficial, the process of discovering new filters to constrain the environment must be improved. There is a close relationship between environment generation in the filter-based approach and abstraction refinement as used during conservative abstractions (e.g. predicate abstraction as discussed in the next section): in both cases one starts with an over-approximation of system/environment behaviors and guided by counterexamples one eventually creates a sufficiently precise system/environment for analysis. Automating this process has received a great deal of attention in abstraction refinement [3, 39, 4, 17], but similar approaches in environment generation are still lacking. Due to the close relationship between these two areas, recent improvements in abstraction refinement should be investigated in the context of environment generation. We have also begun to investigate methods for automatically synthesizing environments of software components, such that the components satisfy given properties [19].

Our view of environment generation is from the perspective of a stand-alone verification activity, with people, possibly other than the software developers, doing the analysis by model checking. However, it can also be viewed from the perspective of integrating model checking with traditional testing activities. In this case, the environment could be constructed by modifying an existing test-harness. However, the technique of using nondeterminism to over-approximate the environment is a paradigm shift from traditional testing which is based on explicit test sequences. Our experience with allowing developers to create environments for model checking is that they are inclined to use the same environment as for testing, and hence do not exploit the ability of the model checker to automatically explore the environment input/response choices in addition to the scheduling choices. Therefore, for model checking to work in practice, it may be necessary to develop methods for generalizing or converting test drivers or test cases into verification environments.

Acknowledgments

Klaus Havelund contributed the content of the Promela overview; any errors are due to our severe editing. We thank Phil Oh, Robert Goldman, Klaus Havelund, Charles Pecheur, Michael Lowry, Thomas Uribe, Hassen Saidi, Matt Dwyer, John Hatcliff, David Dill, Satyaki Das, Jens Skakkabaek, Darren Coffer, Murali Rangarajan, Dimitra Giannakopoulou, Flavio Lerda, Alex Groce, Oksana Tkachuk, Cindy Kong for numerous technical discussions that contributed to this work. We also thank the many reviewers who have provided comments on this work as it progressed. This work was funded by the NASA Information Technology Base Research Program, with follow-on support from the Computing, Information and Communication Technology Program and the Engineering for Complex Systems Program, all supported by NASA's Office of Aerospace Technology.

References

1. R. Allen, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proc. 6th SIGSOFT FSE*, Lake Buena Vista, Florida, November 1998. ACM.
2. J. M. Atlee and J. Gannon. State-based model checking of event-driven systems requirements. *IEEE TSE*, 19(1):24–40, January 1993.
3. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. of TACAS 2001*, volume 2031 of *LNCS*, Genova, Italy, April 2001. Springer-Verlag.
4. T. Ball, A. Podelski, and S. K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proc. of TACAS 2002.*, volume 2280 of *LNCS*, Grenoble, France, April 2002. Springer-Verlag.
5. T. Ball and S. Rajamani. Bebop: A symbolic Model Checker for Boolean Programs. In *Proc. 7th International SPIN Workshop*, volume 1885 of *LNCS*, Stanford University, California, USA, August 2000. Springer-Verlag.
6. B. Beizer. *Software Testing Techniques.* 2nd ed, Van Nostrand Reinhold, New York, 1990.
7. B. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.
8. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
9. D. Bosnacki and D. Dams. Integrating real time into Spin: A prototype implementation. In *Proc. FORTE/PSTV XVIII*, pages 423–439. Kluwer, 1998.
10. E. Brinksma and A. Mader. Verification and optimization of a PLC control schedule. In *Proc. 7th SPIN Workshop*, pages 73–92. Springer-Verlag, 2000.
11. B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems.* Prentice Hall, 2000.
12. R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE TSE*, 19(1):3–12, 1993.
13. J. Falk C. Kaner and H.Q. Nguyen. *Testing computer Software.* 2nd ed, Wiley, 1993.
14. 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 *Proc. 21st International Conference on Software Engineering*, pages 142–151, Los Angeles, May 1999. ACM Press.
15. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE TSE*, 24(7):498–520, July 1998.
16. J.J. Chilenski and S.P. Miller. Applicability of modied condition/decision coverage to software testing. *Software Engineering Journal*, 9(5), Sep 1994.
17. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based Abstraction-Refinement using ILP and Machine Learning Techniques. In *Proc. 14th*

Conference on Computer-Aided Verification, LNCS. Springer-Verlag, July 2002.

18. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.
19. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*. Springer-Verlag, April 2003.
20. M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. 10th Conference on Computer-Aided Verification*, volume 1427 of *LNCS*. Springer-Verlag, July 1998.
21. J. Corbett. Constructing compact models of concurrent Java programs. In M. Young, editor, *Proc. Intl. Symposium on Software Testing and Analysis*, Software Engineering Notes, pages 1–10. SIGSOFT, ACM, March 1998.
22. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. 22nd Intl. Conf. on Software Engineering*. ACM Press, June 2000.
23. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag.
24. Z. Dang and R. Kemmerer. Using the ASTRAL Model Checker to Analyze Mobile IP. In *Proc. IEEE 21st International Conference on Software Engineering*, pages 132–141, Los Angeles, May 1999. ACM Press.
25. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Proc. International Conference on Conputer-aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 160–171. Springer-Verlag, 1999.
26. C. Demartini, R. Iosif, and R. Sist. A deadlock detection tool for concurrent Java programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
27. C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proc. 6th SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
28. N. Dor, M. Rodeh, and S. Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *Workshop on Program Analysis For Software Tools and Engineering*, pages 27–34. ACM, 1998.
29. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proc. 23rd International Conference on Software Engineering*, Toronto, Cananda., May 2001. ACM Press.
30. M. Dwyer and C. Pasareanu. Filter-based model checking of partial systems. In *Proc. 6th ACM SIGSOFT FSE*. ACM SIGSOFT, November 1998.
31. D. Evans. Static detection of dynamic memory errors. In *Conference on Programming Language Design and Implementation*, pages 44–53. ACM, 1996.
32. P. Godefroid. Model checking for programming languages using Verisoft. In *Symp. on Principles of Programming Languages*, pages 174–186. ACM, 1997.
33. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Vericifaction*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.
34. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. Intl. Symp. on Software Testing and Analysis*. ACM Press, July 2002.

35. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal analysis of the remote agent before and after flight. In *5th NASA Langley Formal Methods Workshop*. NASA, 2000.
36. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Intl. Journal on Software Tools for Technology Transfer*, 1999.
37. K. J. Hayhurst, C. A. Dorsey, J. C. Knight, N. G. Leveson, and G. F. McCormick. Streamlining software aspects of certification: Report on the SSAC survey. Technical Report NASA/TM-1999-209519, NASA Langley Research Center, 1999.
38. C. Heitmeyer. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE TSE*, 24(11):927–948, nov 1998.
39. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. Symp. on Principles of Programming Languages*, pages 179–190. ACM, 2002.
40. G. Holzmann. The model checker SPIN. *IEEE TSE*, 23(5):279–295, 1997.
41. G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE TSE*, 28(4):364–377, April 2002.
42. G.J. Holzmann. Logic verification of ansi-c code with spin. In *Proc. 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 131–147. Springer Verlag, Sep. 2000.
43. G. Hwang, K. Tai, and T. Hunag. Reachability testing: An approach to testing concurrent software. *Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.
44. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In Mary Jean Harrold, editor, *Proc. International Symposium on Software Testing and Analysis*, Software Engineering Notes, pages 14–25, Portland, Oregon, August 2000. ACM Press.
45. JPL Special Review Board. Report on the loss of the Mars Polar Lander and Deep Space 2 missions, March 2000.
46. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*. Springer-Verlag, April 2003.
47. R. Lutz. Analyzing software requirements errors in safety-critcal embedded systems. In *Proc. IEEE International Symposium on Requirements Engineering*. IEEE Computer Society, January 1993.
48. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
49. G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. 21st International Conference on Software Engineering*, pages 399–410. ACM Press, May 1999.
50. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proc. 22nd International Conference on Software Engineering*. ACM Press, June 2000.
51. A. Pnueli. The Temporal Logic of Programs. In *18th annual IEEE-CS Symposium on Foundations of Computer Science*, pages 46–57, 1977.
52. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer.
53. PolySpace. `http://www.polyspace.com`.

54. C. Păsăreanu, M. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Proc. 6th SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
55. C.S. Păsăreanu, M.B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of TACAS 2001*, volume 2031 of *LNCS*. Springer-Verlag, 2001.
56. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*. Springer-Verlag, 1982.
57. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC 9/FSE 10*, pages 267–276, Sep 2003.
58. RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-178B, RTCA, Inc., dec 1992.
59. J. Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.
60. H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *Proc. 14th IEEE International Conference on Automated Software Engineering*, pages 92–101. IEEE Computer Society, October 1999.
61. H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc. 11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454. Springer-Verlag, July 1999.
62. Sha, Klein, and J. Goodenough. Rate monotonic anaysis for real-time systems. *Foundations of Real-Time Computing*, pages 129–155, 1991.
63. G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.
64. Microsoft Spec and Check Workshop, 2001. `http://research.microsoft.com/specncheck/`.
65. S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, August 2000.
66. S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time. In *Proc. of TACAS 1996*, volume LNCS 1055. Springer, 1998.
67. M. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, pages 238–266. LNCS, 1043, Springer Verlag, 1996.
68. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 14th IEEE International Automated Software Engineering Conference*. IEEE Computer Society, September 2000.
69. W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce objected-oriented programs for model checking. In Mats P. E. Heimdahl, editor, *Proc. Third ACM Workshop on Formal Methods in Software Practice*, pages 3–12, Portland, Oregon, August 2000. ACM Press.
70. C. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *International Symposium on Software Testing and Analysis*, pages 153–162. ACM Press, 1998.