USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

Using Model Checking to Debug Device Firmware

Sanjeev Kumar*
Department of Computer Science
Princeton University
skumar@cs.princeton.edu

Kai Li
Department of Computer Science
Princeton University
li@cs.princeton.edu

Abstract

Device firmware is a piece of concurrent software that achieves high performance at the cost of software complexity. They contain subtle race conditions that make them difficult to debug using traditional debugging techniques. The problem is further compounded by the lack of debugging support on the devices. This is a serious problem because the device firmware is trusted by the operating system.

Model checkers are designed to systematically verify properties of concurrent systems. Therefore, model checking is a promising approach to debugging device firmware. However, model checking involves an exponential search. Consequently, the models have to be small to allow effective model checking.

This paper describes the abstraction techniques used by the ESP compiler to extract abstract models from device firmware written in ESP. The abstract models are small because they discard some of the details in the firmware that is irrelevant to the particular property being verified. The programmer is required to specify the abstractions to be performed. The ESP compiler uses the abstraction specification to extract models conservatively. Therefore, every bug in the original program will be present in the extracted model.

This paper also presents our experience with using Spin model checker to develop and debug VMMC firmware for the Myrinet network interfaces. An earlier version of the ESP compiler yielded models that were too large to check for system-wide properties like absence of deadlocks. The new version of the compiler generated abstract models that were used to identify several subtle bugs in the firmware. So far, we have not encountered any bugs that were not caught by Spin.

1 Introduction

Device firmware has to be reliable because it is trusted by the operating system. It has the ability to write directly into the physical memory. A stray memory write resulting from a bug can corrupt critical data structures in the operating system and crash the entire machine.

Writing reliable firmware for devices is a challenging problem for three reasons. First, the firmware is implemented using concurrency [25]. And concurrent programs are inherently hard to write correctly. Often, they have unforeseen interactions between the different sequential flows of control resulting in race conditions. Second, event-driven state machines are used to express the concurrency because of their low performance overhead. However, programming with event-driven state machines in languages like C is difficult because they are not designed to support event-driven state-machines programming. Event-driven state-machines programs can be written in these languages using an explicit interface [25] which requires state machines to be specified explicitly using function pointers. The resulting programs are difficult for the programmer to understand and for the compiler to compile efficiently. To get good performance, the programmer is forced to perform some optimizations manually. This introduces subtle bugs in the program. Third, very limited debugging support is available on the devices. Often it is limited to a few memory locations to which the device can write. The programmer has to diagnose the bugs by observing these memory locations on the host machine.

The earlier version of the Virtual Memory-Mapped Communication (VMMC) firmware [14] for Myrinet network interface cards[1] was implemented using event-driven state machines in C. Since the VMMC architecture [5] delivers high-performance on gigabit networks by migrating as much functionality as possible from the operating system to the network interface card, the network interface firmware is fairly complex. Our experience with implementing the VMMC firmware in C has been that while good performance could be achieved, the source code was difficult to write, maintain, and debug. Even after several man-years of debugging, bugs due to race conditions remain and occasionally cause system crashes.

Model checking is a promising approach to building reliable firmware. Model checkers take a model of the system and explore all possible interleaved executions of the concur-

*Now at Microprocessor Research Labs, Intel Corp.

[1] The network interface card has a programmable 33-MHz LANai4.1 processor, 1-Mbyte SRAM memory.

rent system. Since the number of possible executions grows exponentially with the size of the model, abstract models that hide details in the original system are necessary. In addition, often only a fraction of the model can be explored. In spite of these limitations, the systematic search performed by the model checker results in much more extensive testing than traditional methods.

ESP [25] is a language for writing firmware for programmable devices. It uses a model checker to aid in developing and debugging the programs. The language is designed so that its compiler can extract models that can be used by a model checker like Spin [21] to debug the firmware. In the software community, model checking has traditionally been used to find hard-to-find bugs in working systems [7, 15, 27, 30, 18, 17, 22, 11]. In contrast, Spin is used throughout the firmware development process. Usually, the program is developed and debugged entirely using Spin before it is ever run on the device. This is because developing firmware on the device is a slow and painstaking process

Since the version of the VMMC firmware implemented in C was buggy, the firmware was reimplemented using ESP [25]. The ESP compiler extracted models that were very useful in implementing the firmware. A model was used to develop and debug a retransmission protocol in the firmware. It was also used to verify memory safety in the firmware. However, in both of these cases, the models were small because the properties being verified were local and involved only a few ESP processes. The model extracted by this version of the ESP compiler was too big to check for system-wide properties like the absence of deadlocks in the firmware. Since the system-wide bugs are especially difficult for the programmer to find precisely because they are nonlocal, this was a significant limitation of that compiler.

This paper presents techniques used by the new version of the ESP compiler that extracts abstract models. Instead of generating a single model, the compiler now extracts several different models depending on the property that is being checked. These abstract models are significantly smaller because they omit (i.e. abstract away) certain details in the ESP program that are not relevant to the property being verified. This paper also presents our experience with using abstract model to find deadlock bugs in the VMMC firmware. Our main conclusions are as follows:

- The compiler can be used to extract conservative abstract models. In ESP, the abstractions are specified by the programmer. The compiler uses these abstractions conservatively to generate models. Therefore, even if a programmer makes a mistake in specifying the abstraction, every bug in the program will be present in the model. The novelty of this approach is that it gives the programmer control over the abstraction process without relying on the programmer to be correct.

- Abstraction was necessary to generate models that could be used to check for system-wide properties in the VMMC firmware. Using the abstract models, the model checker uncovered seven bugs that would cause the firmware to deadlock. These were subtle bugs that were not caught even after careful code inspection and months of testing and debugging.

- Partial explorations by model checkers can be very effective for debugging. Even using the abstract models, Spin could not exhaustively check the VMMC firmware for deadlocks because of resource constraints. However, in ESP, the model checker is meant to be used as a debugging tool and not to certify correctness. A partial exploration by the model checker uncovered the seven bugs mentioned in the previous paragraph. In addition, we have not encountered any bugs (that were not caught by Spin) while running the firmware on the device.

The rest of the paper is organized as follows. Section 2 presents a brief description of model checkers. Section 3 presents our approach. Section 4 discusses the techniques used by the compiler to generate tractable models. Section 5 describes our experience with using the Spin model checker to develop and debug VMMC firmware. Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 Model Checking

Model checking is a technique for verifying a system composed of concurrent finite-state machines. Given a concurrent finite-state system, a model checker explores all possible interleaved executions of the state machines and checks if the property being verified holds. A *global state* in the system is a snapshot of the entire system at a particular point in execution. The *state space* of the system is the set of all the global states reachable from the initial global state. Since the state space of such systems is finite, the model checkers can, in principle, exhaustively explore the entire state space.

Model checkers can check for a variety of properties. These properties are traditionally divided into *safety* and *liveness* properties. Safety properties are properties that have to be satisfied in specific global states of the system. Assertion checking and deadlock are safety properties. Assertions are predicates that have to hold at a specified point in one of the state machines. This corresponds to the set of global states where that state machine is at the specified point and the predicate holds. A deadlock situation corresponds to the set of all the global states that do not have a valid next state. Liveness properties are ones that refer to sequence of states. Absence of livelocks is a liveness property because it corresponds to a sequence of global states where no useful work gets done. Liveness properties are specified using temporal logic.

The advantage of using model checking is that it is automatic. Given a specification for the system and the property to be verified, model checkers automatically explore the state space. If a violation of the property is discovered, it can produce an execution sequence that causes the violation and thereby helps in finding the bug.

There are two problems with using model checkers. First, the state space to be explored is exponential in the number of processes and the amount of memory used. So the resources required (CPU as well as memory resources) by the model checker to explore the entire state space can quickly grow beyond the capacity of modern machines. Second, the specification language supported by the model checkers provides limited functionality. So, it is not straightforward to translate concurrent programs written in traditional programming languages into the specification language of the model checkers.

Abstraction is the key to addressing both these problems. Depending on the property be verified, a model that captures only the details relevant to that property needs to be extracted. For properties involving small subsystems, detailed models can be used. However, for properties involving large subsystems, abstract models have to be used.

Models are usually extracted by hand. This process can be time consuming. In addition, it is hard to be sure that the model accurately captures the actual system. Worse yet, as the system evolves, the model has to be independently updated to reflect the changes. Therefore, the use of model checkers is greatly simplified when the models can be extracted automatically[22, 11].

3 Our Approach

Event-driven State-machine Programming (ESP) [25] is a language for programming devices. ESP adopts several structures from the CSP [19] language and has C-style syntax. The basic components of the language are processes and channels. Each process represents a sequential flow of control in a concurrent program and communicates with other processes using rendezvous channels.

ESP is designed to meet three goals. First, ESP should provide language support that makes it easier to develop device firmware. Second, it should allow the use of model checkers like Spin [21] to extensively test and debug the firmware. Third, the compiler should be able to generate efficient executables to run a single processor.

In traditional languages like C, event-driven state-machines programs can achieve high performance by giving up ease of development and reliability. Therefore, they meet only one of the three goals.

To meet all three design goals, the ESP language is designed so that it can not only be used to generate an executable but also be translated into models that can be used by the Spin model checker (figure 1). The ESP compiler takes an ESP program (`pgm.ESP`) and generates 2 types of files. The generated C file (`pgm.C`) can then be compiled together with the C code provided by the programmer (`help.C`) to generate the executable. The programmer-supplied C code implements simple system-specific functionality like accessing device registers to check for network message arrivals. The Spin files (`pgm[1-N].SPIN`) generated by the ESP compiler can be used together with programmer-supplied Spin code (`test[1-N].SPIN`) to verify different properties of the system. The programmer-supplied Spin code generates external events such as network message arrival as well as specifies the properties to be verified.

For each property to be verified, the programmer has to provide test code written in Spin (test[1-N].SPIN in Figure 1). This code is usually fairly small (around 100 lines). Once the test code is written, it can be used to check the ESP program for bugs every time the program is modified.

The earlier version of the ESP compiler [25] generated a single Spin model that included all the details in the ESP program. However, while these models were very useful for checking properties of subsystems consisting of 1-2 processes, they could not be used to check for system-wide properties such as absence of deadlocks. This was due to state-space explosion. Since the hard-to-find bugs are often due to race conditions involving several different processes, ESP now supports automatic extraction of abstract models. Using an abstract model (Section 4), we were able to check for system-wide deadlocks. We found several bugs that resulted in deadlocks (section 5.1).

The ESP approach differs from the previous efforts as follows:

Domain-specific Language ESP is designed not only to simplify the task of programming devices but also to make it easier to extract models. Consequently, any detail in the ESP program that is necessary to check a particular property can be retained in the extracted model. In contrast, general-purpose languages like C, C++, and Java have language features (complex pointer manipulation, exceptions etc.) that are difficult to translate into the specification language of the model checkers [18, 22, 11, 26].

Support for Abstraction Other domain-specific languages [8, 3, 4] extract a single model from the program and use it for model checking. To avoid the state-space explosion associated with detailed models, these languages have been designed to encode only the control structure of the program—the data manipulation is implemented externally in C. In contrast, the ESP language provides support for both control structure and data manipulation. The ESP compiler uses new abstraction techniques to discard some unnecessary details and generate more tractable models.

This paper presents the abstraction techniques used by the new version of the ESP compiler. These techniques are general enough to be applicable to general-purpose languages like C and Java. However, the design of the ESP language makes them particularly effective on ESP programs. For instance, each object in an ESP program can be pointed to by only one of the processes in the program. This limits the amount of pointer aliasing that can occur. Consequently, the increase in state space during abstraction due to aliasing is small (Section 4.2).

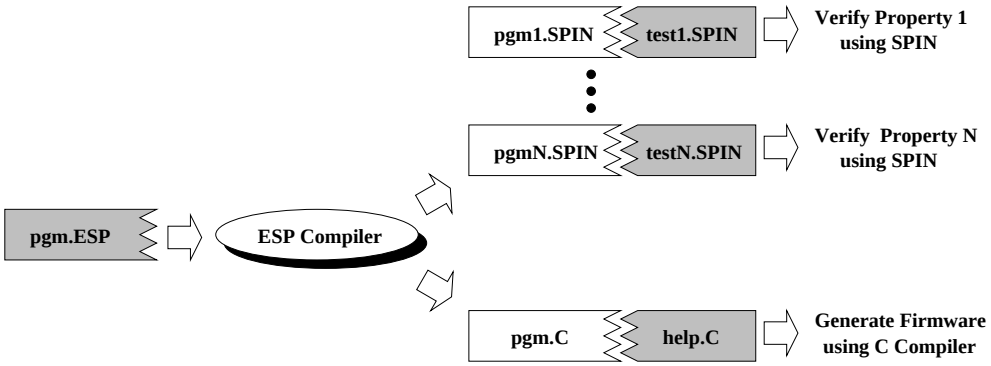

Figure 1: The ESP approach. The shaded regions represent code that has to be provided by the programmer.

3.1 Spin Model Checker

Currently, ESP used the Spin model checker [21]. Spin is a flexible and powerful model checker designed for software systems. Spin supports high-level features like processes, rendezvous channels, arrays, and records. Most other model checkers target hardware systems. Although ESP can be translated into these languages, additional state would have to be introduced to implement features like the rendezvous channels using primitives provided in the specification language. This would make the state explosion problem worse. In addition, the semantic information lost during translation would make it harder for the model checker to optimize the state-space search. Spin allows verification of safety as well as liveness properties.

Spin is a on-the-fly model checker and does not build the global state machine before it can start checking for the property to be verified. So, in cases where the state space is too big to be explored completely, it can do partial searches. It provides 3 different modes for state-space exploration. The entire state space is explored in the *exhaustive* mode. For larger systems, the *bit-state hashing* mode performs a partial search using significantly less memory. It exploits the fact that state spaces are usually sparse and uses a hash function to obtain a much more compact representation for a state. However, since the hash function can map two states onto the same hash, a part of the state space may not be explored. This technique often allows very high coverage ($> 98\ \%$) while using an order of magnitude less memory. The *simulation* mode explores single execution sequence in the state space. A random choice is made between the possible next states at each stage. Since it does not keep track of the states already visited, it requires very little memory. However, it could explore some states multiple times while never exploring some other states.

4 Extracting Models Using a Compiler

The ESP compiler generates three types of models: *detailed*, *memory-safety*, and *abstract*. The detailed models contain all the details from the original ESP program. These detailed models often have too much state to be able to perform effective state-space exploration. However, these models are useful during the development and debugging of the firmware using the simulation mode in Spin. They can also be used to check for properties in small subsystems.

The memory-safety models generated are used to check for memory allocation bugs in the program. These models are essentially detailed models with some additional Spin code inserted to check for validity of memory accesses. They contain even more state than the detailed models. In spite of this, these models can be usually used to exhaustively explore the state space for allocation bugs. This is because the memory safety of each individual process can be checked separately using the model checker.

The abstract models generated by the ESP compiler omit some of the details that are irrelevant to the particular property being verified. These models can have significantly smaller state than the detailed models. These models can be used to check for system-wide properties like absence of deadlocks. This class of bugs usually involves several different processes. These bugs are especially hard to find.

Earlier papers [25, 23] described how detailed and memory-safety models are extracted from ESP programs by the ESP compiler.[2] Extracting abstract models requires additional techniques that allow it to abstract away some irrelevant details in the ESP program. These techniques are described in the rest of this section. The simple ESP program in Figure 2 will be used to illustrate the model-extraction process.

In ESP, the programmer specifies the abstractions. The advantage of this approach is that it gives the programmer control over the abstraction process. It allows the programmers to use their understanding about the program and the property being verified to choose the appropriate abstraction. This can result in a better abstraction than ones that can be obtained through static analysis by a compiler.

The ESP compiler uses the abstractions specified by the programmer conservatively when generating the abstract models. Consequently, a bug in the ESP program will always

[2] Briefly, ESP processes and channels are translated into Spin processes and synchronous channels respectively. Since Spin does not support pointers and dynamic allocation while ESP supports them, ESP objects cannot be directly be translated into Spin objects. However, the details of the translation are not necessary for following the rest of this paper.

be present in the abstract model. Our approach is based on a well-known technique[3] of using nondeterminism to broaden the scope of model checking [21, 11, 10]. However, earlier efforts have focused on abstracting simple types like integers. This paper demonstrates how these techniques can be extended to handle complex data types like records, unions, and arrays. This requires addressing additional problems that arise due to pointer aliasing.

4.1 Specifying Abstractions

The abstractions to be performed by the compiler have to be specified by the programmer. The ESP compiler currently allows the programmer to specify two types of abstractions.

Replacing Types. It allows a complex type to be replaced by a much simpler type. This can be done either by specifying an alternative type for the variables individually or by specifying an alternative type in a type declaration. For instance, if the original program contained the following type declaration:

```
type replyT = record of {
  caller: int,
  last: bool,
  addr: int,
  size: int
}
```

then the programmer can specify the following abstraction:

```
replace type replyT = record of {
  caller: int,
  last: bool
}
```

Currently, ESP requires the replacement type to be a supertype of the original type. Essentially, it allows fields from records and unions to be dropped.

Replacing a complex type by a simpler type can significantly reduce the amount of state in the model. For instance, the code to implement the retransmission protocol accepts packets that are implemented as a union of the different types of packets that have to be sent. However, as the content of the packet makes little difference to the correctness of the retransmission code, the complex datatype representing packets can often be replaced by a simple type in the abstract model. Another simplification that can reduce the amount of state is to use smaller arrays than used in the original program. Often, the size of arrays affects only the performance and not the correctness of the program.

Dropping Variables. Some variables that do not affect the validity of a property being checked can be dropped altogether. For instance, a table that keeps track of the mapping between virtual and physical addresses in the main memory might not be relevant when checking the firmware for deadlocks. The variable `table` in process `pageTable` can be dropped by specifying the following abstraction:

```
drop pageTable $table
```

4.2 Extracting Abstract Models

The ESP compiler uses programmer-specified abstractions to generate abstract models. First the compiler performs a type-checking phase during which the compiler determines a type for every expression in the original program (without taking the abstractions into account). The model generator phase can apply the abstractions to each of the statements of this fully-typed program independently.

The abstractions specified can cause some of the expressions in a statement to have an indeterminable value. In these situations, the ESP compiler uses nondeterminism to make conservative approximations that strictly generalizes the scope of model checking. The various expressions in a statement can be classified into two classes: *left-exp* and *right-exp*. They are handled as follows:

left-exp. A left-exp is an expression that is used to determine a memory location to which a value will be stored. These expressions appear on the left side of the assignment statements and in `in` operations on channels. Consider the following statements:

```
a = b;
a[i].last = d;
```

where variable `a` has the type

```
type tableT = #array of #record of {
  first: int, last: int
}
```

In the simplest cases, when a left-exp becomes undeterminable, the statement can be simply discarded during model extraction. For instance, if the variable `a` is dropped by the abstraction, the first statement `a = b;` becomes irrelevant and can be discarded. This is because the only side effect of that statement is to the variable `a`. Similarly, if the `last` field is dropped from the type `tableT`, the second statement can be discarded during model extraction. This is because all objects of that type no longer have the `last` field. As a result, the statement has no remaining side effect in the generated model.

The most general case that has to be handled occurs in the second statement when variable `a` or variable `i` is dropped.

[3] When performing abstractions, some of the values in the model might become *undeterminable*. For instance, the value of a variable in the model that depended on another variable in the original program that was discarded during abstraction will become undeterminable. The compiler keeps track of these values and makes sure that the abstract model only broadens the scope of model checking. For example, when the value of a condition in a conditional statement cannot be determined, it can be replaced by a nondeterministic choice in the model. During model checking, both the branches of the conditional statement will be explored. Broadening the scope of model checking can introduce spurious bugs (false positives) in the abstract model. However, all the bugs that were present in the program will be retained in the model.

```
#define TABLE_SIZE     100
#define PAGE_SIZE      4096
#define PAGE(a)        ( (a) / PAGE_SIZE)
#define OFFSET(a)      ( (a) % PAGE_SIZE)
#define ADDR(p)        ( (p) * PAGE_SIZE)

type reqT = record of { caller: int, addr: int, size: int}
type replyT =  record of { caller: int, last: bool, addr: int, size: int}
channel reqC : reqT
channel replyC : replyT

process pageTable {
  $table : #array of int = #{ TABLE_SIZE ~> 0 ...};
  // Omitted : Code to initialize the table
  while ( true) {
    in( reqC, { $caller, $vaddr, $size});   // Get the next request
    assert( !OFFSET(vaddr));      // Assumes vaddr is page aligned
    $done: bool = false;
    while( !done) {
      $paddr : int = ADDR(table[PAGE(vaddr)]);  // Lookup physical address
      $chunk : int = PAGE_SIZE;     // Calculate the size
      if ( size < PAGE_SIZE)    chunk = size;
      size = size - chunk;
      done = ( size == 0);
      out( replyC, { caller, done, paddr, size}); // Send a reply
    }
  }
}

process transfer {
  while ( true) {
    // Omitted : Code that generates values for variables 'vaddr' and 'size'
    out( reqC, { @transfer, vaddr, size});
    $last : bool = false;
    while( !last) {
      in( replyC, { @transfer, last, $paddr, $chunk});
      // Omitted : Code to transfer 'chunk' bytes at 'paddr'
    }
  }
}
```

NOTES

1. Process `pageTable` translates virtual addresses into physical addresses. It maintains a table that maps virtual page numbers into physical page numbers. It accepts translation requests on channel `reqC` and sends replies on channel `replyC`. Since a region of contiguous virtual memory can map onto a set of noncontiguous physical pages, each request sent on channel `reqC` can yield multiple replies on channel `replyC`. The last reply is identified by the `last` field in the reply.
2. Process `transfer` computes a pair of `vaddr` and `size` that identifies a region in virtual memory. It sends a request on `reqC` to translate it into physical addresses. It then receives the physical addresses on channel `replyC` and uses it to transfer data.
3. Since other processes might be sending requests on channel `reqC`, the caller field on the channels `reqC` and `replyC` is used to match the replies with the request. `@transfer` is a constant that represents the process id for the process `transfer`.

Figure 2: A ESP program.

In this case, the object pointed to by `a[i]` is being mutated, and the change would be visible to any other pointer that was pointing to the same object. To handle this case, the compiler has to determine a list of pointers to which `a[i]` could be aliased. For each of these pointers, the generated model has to include a nondeterministic statement that either updates the object to which it points or does not update that object.

Nondeterministically updating a large set of objects can dramatically increase the amount of state space that has to be explored. It can also result in false-positive bugs being introduced into the model. A number of techniques can be used to narrow the list of pointers to which `a[i]` can be aliased. First, only pointers of the same type as `a[i]` have to be considered. Second, only pointers within the same process can be aliased to `a[i]`, since processes in ESP do not share objects. Third, in the case where only variable `i` is dropped, only objects pointed to by an entry in array `a` needs to be considered. Finally, alias analysis can be used to further reduce the list of pointers. If compile-time analysis can determine that the pointer `a[i]` is not aliased to any other pointer, the situation reduces to the simple case where the statement is simply discarded.

Pattern matching also needs to be performed conservatively. We will illustrate this using code shown in Figure 3. In ESP, a pattern can appear on the left-hand side of an assignment statement or in an `in` statement. For instance, the `in` operation on channel `patternC` in process `A` uses a pattern. The pattern specifies that it expects the value `5` in the `caller` field of the record it receives on the channel. Consequently, the pair of `in` and `out` operations will succeed only if the `out` operation in process `B` supplies the value `5` in the `caller` field.

Suppose the programmer uses the abstraction in Figure 3 on the ESP program in the same figure. The abstraction drops the `caller` field on the channel. In the absence of the `caller` field in the extracted model, there is no way to determine if the pattern matching on the channel should succeed. Therefore, a nondeterministic statement is inserted in process `A` before the `in` operation on the channel. The `in` operation will succeed only if the variable `match` is set to `OK`. This captures both cases: the case in which the `in` operation on channel `patternC` would have succeeded and the case in which it would not have succeeded.

Being conservative on patterns ensures that a deadlock state is represented in the extracted model. A deadlock state could get translated into a live state if an exit transition is added to that state during model extraction. ESP avoids this by ensuring that each state in the program in which a process could be blocked on a particular channel is represented by a state in the extracted model where that process is blocked on that channel.

right-exp. All expressions that are not left-exp expressions are right-exp expressions. These generate values to be used at various points in the programs. They appear on the right side of the assignment statements, in conditionals of `if` and `while` statements, and in `out` operations on channels.

During abstraction, the value of a right-exp expression can become undeterminable. Ideally, these expressions should be replaced by one that nondeterministically returns a valid value of the type of the expression. This will cause the model checker to try all possible valid values during the state space exploration.

For boolean expressions, only two values are possible and a nondeterministic choice between the two can be made. So, a boolean expression in a conditional statement (like an `if` statement) whose value can no longer be computed is replaced by a nondeterministic statement [21]. During model checking, both the branches of the conditional statement will be explored.

ESP Program

```
type patternT = record of {
  int caller, int count
}
channel patternC : patternT
process A {
  in( patternC, { 5, $count});
}
process B {
  // Omitted: code to declare and initialize 'caller'
  out( patternC, { caller, 45});
}
```

Abstraction

```
replace type patternT = record of {
  int count
}
```

Extracted Spin Model

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef patternT { int nil; int count; };
chan patternC = [0] of { mtype, patternT };
proctype A() {
  mtype match; int count;
  // Nondeterministically pick a value for 'match'
  if
  :: skip -> match = OK
  :: skip -> match = DONT_RECV
  fi;
  patternC ? eval(match), 0, count
}
proctype B() {
  patternC ! OK, 0, 45
}
```

Figure 3: A program to illustrate the handling of patterns. For simplicity, some details that are unnecessary to understand this example have been omitted.

For nonboolean expressions, trying all possible valid values would be computationally very expensive during model checking. It is also usually unnecessary because a small set of values can effectively cover the entire space. However, there is no general way for the ESP compiler to determine the set of values that would be sufficient to cover the entire state space. Therefore, the ESP compiler relies on the programmer to supply the right set of values. For each variable in the program (except boolean variables) for which the abstract model needs a nondeterministic value, a channel is generated in the abstract model. When a value is needed, the model performs a read operation on the channel. The programmer is responsible for supplying values on the channel using a nondeterministic statement. For instance, the following code executes an infinite `do` loop and nondeterministically supplies either of the three values (5, 6, and 9).

```
do
:: intC ! 5
:: intC ! 6
:: intC ! 9
od
```

For nonscalar types like arrays and records, the set of values that the programmer has to provide on these channels not only includes new objects but also all existing objects in the model to which it could be aliased.

4.3 Discussion and Limitations

Size of State Space. In principle, an abstract model generated can have more states than the corresponding detailed model. This is because two different things happen during abstraction. First, some details in the program are discarded. This will reduce the number of states. Second, the compiler uses the abstractions specified conservatively by translating some of the deterministic statements in the program into nondeterministic statements. This can increase the number of states that have to be explored.

In practice, the abstract model has significantly fewer states. This is because only a small number of nondeterministic statements get introduced. In addition, since the programmer specifies the abstraction in ESP, the programmer can pick the abstraction carefully so as to minimize the number of states. For instance, if the packet sequence number in the implementation of a network retransmission protocol is unnecessary to verify a particular property, the programmer can specify that all variables and fields of records that store sequence numbers in the program should be discarded from the abstract model. Then, the model will no longer have any statement that uses the sequence number. Consequently, no new nondeterministic statements will be introduced into the model due to this abstraction.

Bugs. By being conservative, ESP ensures that all bugs in the original program are present in the extracted model. However, this does not guarantee that all bugs will be caught during model checking. First, the model checker might not be able to check the entire state space because of resource constraints. Second, ESP relies on the programmer to provide test code (Section 3) as well as code that generates values on some channels (Section 4.2). A mistake by the programmer can result in bugs being missed during model checking.

The techniques described in this paper help reduce and isolate the portion of Spin code where mistakes can be made by a programmer. This is analogous to type-safe languages that rely on C to implement unsafe portions of a program. In this case, a program can have type-safety errors. However, these errors are isolated in the portion of the program that is implemented in C.

Abstraction 1

```
replace type reqT = record of {
  caller: int
}
replace type replyT = record of {
  caller: int,
  last: bool
}
drop pageTable $table, $vaddr, $size
drop pageTable $paddr, $chunk
drop transfer $vaddr, $size, $paddr
drop transfer $chunk
```

Abstraction 2

```
replace type reqT = record of {
  caller: int
}
replace type replyT = record of {
  caller: int
}
drop pageTable $table, $vaddr, $size
drop pageTable $paddr, $chunk, $done
drop transfer $vaddr, $size, $paddr
drop transfer $chunk, $last
```

Figure 4: Two Abstractions for the ESP program in Figure 2

4.4 Example

In this section, we will use an example to illustrate the use of abstraction to check for a property. We will check for absence of deadlocks in the ESP program in Figure 2.

'Abstraction 1' in Figure 4 can be used to check the program for the absence of deadlocks. Using the abstraction, the ESP compiler generates an abstract model (Figure 5). The abstraction drops all variables except `caller` and `done` in process `pageTable` and `last` in process `transfer`. It also replaces the types of the two channels. During abstraction, the value of the boolean variable `done` becomes indeterminate because its value depends on the value of the variable `size` that was dropped. The compiler translates the state-

ment

```
done = ( size == 0);
```

into Spin code that nondeterministically assigns either values `true` or `false` to it as follows:

```
if
:: skip -> done = 0
:: skip -> done = 1
fi
```

The Spin model checker can exhaustively explore the entire state space (12 states!) and determine that there are no deadlocks. In contrast, if a detailed model was used, the model checker would have to potentially explore a large number of states (by trying all possible values for `vaddr` and `size`) to determine that there were no deadlocks.

Since the compiler makes conservative approximations when generating abstract models, the model checker will not miss a deadlock because of a programmer error in specifying an abstraction. However, a programmer error can cause a spurious deadlock to be flagged. For instance, 'Abstraction 2' in Figure 4 will detect a spurious deadlock because the programmer dropped the variable `done` by mistake. These will have to be double checked by the programmer.

Finally, we can introduce a deadlock in the program by replacing the line

```
$done: bool = false;
```

by the line

```
$done: bool = ( size == 0);
```

This will cause a deadlock if the `size` specified on channel `reqC` is 0. The model checker will find the bug using either of the two abstractions.

5 Debugging VMMC Firmware

The VMMC firmware was reimplemented using ESP. The earlier implementation in C includes about 15600 lines of C code. In contrast, the new implementation using ESP required 500 lines of ESP code together with around 3000 lines of C code. The C code implements simple tasks like initialization, initiating DMA, packet marshalling and unmarshalling and shared data structures with code running on the host processor (in the VMMC library and the VMMC driver). All the complex state machine interactions are restricted to the ESP code, which uses 8 processes and 19 channels. This is a significant improvement over the earlier implementation where the complex interactions were spread throughout the 15600 lines of hard-to-read code.

The previous version the compiler was used to extract detailed and memory-safety models that were useful in checking for local properties involving 1-2 processes [25, 23]. First, Spin was used throughout the development process. In the software community, model checking has traditionally been used to find hard-to-find bugs in working systems. However, since developing firmware on the network interface card involves a slow and painstaking process, Spin was often used to develop code. For instance, the retransmission code, which uses a simple sliding window protocol with piggyback acknowledgement, was developed and debugged entirely using the Spin simulator. Once debugged, the firmware was ported to the network interface card with little effort. Second, Spin was used to verify memory safety in the VMMC firmware. Instead of supporting safety through garbage collection, ESP provides an explicit `malloc/free`-style interface to support dynamic memory management. Although this interface is unsafe, the Spin model checker can be used to verify memory safety. To facilitate this, ESP uses a novel scheme that makes memory safety a local property of each process. This allows each process to be verified separately resulting in smaller models. Consequently, memory safety of the VMMC firmware could be checked exhaustively.

File	Lines of Code
ESP Program	453
Abstraction Specification	108
Test Code	128
Abstract Model Extracted	2202

Table 1: Sizes (in lines) of the various files used to debug the VMMC firmware. The first three files have to be provided by the programmer while the last one is generated by the ESP compiler.

The size of the models generated by the previous version of the ESP compiler was too large to check for system-wide properties like absence of deadlocks. In the rest of this section, we describe our experience with using the abstract models generated by the new version of the compiler to check for deadlocks in the VMMC firmware.

5.1 Deadlocks in VMMC Firmware

System-wide deadlocks are often a result of complex interactions in the program and can be difficult for programmers to find. Therefore, the use of model checking to find these bugs is important. We used an abstract model to find bugs that would cause the firmware to deadlock.

Table 1 shows the sizes of the various files that were used to find bugs with the abstract model. The abstraction specification specifies 63 variables to be dropped (1 line each in the specification), replaces the type of one variable by a simpler type, and replaces 18 types by simpler types. It is fairly easy for the programmer to identify the parts of the program that should be abstracted away. In the VMMC implementation, only a handful of the variables required closer examination to decide whether or not they needed to be abstracted. The entire abstraction specification took a few hours to write.

Using the abstract model, Spin found several subtle bugs in the VMMC firmware that would cause the firmware to dead-

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef reqT { int nil; int caller; };
typedef replyT { int nil; int caller; bool last; };
chan reqC [NUM_PROCESSES] = [0] of { mtype, reqT };
chan replyC [NUM_PROCESSES] = [0] of { mtype, replyT };

proctype pageTable( int pid) {
  int caller; bool done;
  atomic {
    do
    :: 1 -> {
         reqC[ pid] ? OK, 0, caller;
         done = 0;
         do
         :: ( !done) -> {
              // Nondeterministically assign a value to 'done'
              if
              :: skip -> done = 0
              :: skip -> done = 1
              fi;
              replyC[pid] ! OK, 0, caller, done
            }
         :: else -> break
         od
       }
    od
  }
}
proctype transfer( int pid) {
  bool last;
  atomic {
    do
    :: 1 -> {
         reqC[pid] ! OK, 0, 1;
         last = 0;
         do
         :: ( !last) -> replyC[pid] ? OK, 0, 1, last
         :: else -> break
         od
       }
    od
  }
}
```

Spin NOTES

1. `'?'` and `'!'` are used to receive and send messages on channels. Constants specified in a receive operation has to match the corresponding values in the send operation for the data to be successfully transferred on a channel.
2. `if` and `do` are nondeterministic statements. They differ in that an `if` is executed only once while a `do` is executed repeatedly until a `break` statement is executed in the body. These statements become deterministic (and turn into regular `if` and `do`) statements when there is only one choice or when there are two choices and one of them is an `else` statement.
3. `mtype` is like `enum` in C.
4. `@transfer` in Figure 2 gets translated into the constant `1`.
5. Some details like the `atomic` statement, the `mtype` field, and the `nil` fields can be ignored. They are not needed to understand this example.

Figure 5: Abstract Spin model generated from the ESP program in Figure 2 using 'Abstraction 1' in Figure 4

Spin Search Mode		Exhaustive	Bit-state hashing
Limiting Resource		Memory	CPU Time
No. of States	Stored	117351	22574700
	Matched	265492	77165900
Time (hr:min:sec)		0:01:24	3:57:30
Memory (in MBytes)		268.35	167.92

Table 2: Checking for the absence of deadlocks in the VMMC firmware using Spin. In both Spin modes, the state-space exploration could not be completed because of resource constraints. The *stored* column shows the number of unique states encountered while the *matched* column shows the number of states encountered that had already been visited before.

lock. However, even with the abstract model, an exhaustive search of the state space was not possible because of resource constraints. Therefore, only partial searches were performed. After the bugs that were found were fixed, further state-space exploration using Spin did not uncover any more bugs. Table 2 presents the amount of state space that could be explored using the resources available. In the exhaustive mode, Spin had to abort the search after 84 seconds because it ran out of memory. In the bit-state hashing mode, Spin ran for 3 hours and 57 minutes before the search was terminated by the user.[4] Since Spin only could perform a partial search, we cannot conclude that there are no more bugs in the VMMC firmware. However, we have not encountered any bugs while running the firmware on the device that were not caught by Spin.[5]

Even using a partial search, Spin found seven bugs in the firmware. These were subtle bugs that were not caught even after careful code inspection and months of testing and debugging. The VMMC firmware in ESP was designed to avoid the bugs encountered in the earlier implementation of the firmware in C. In addition, the ESP language allowed the complex interactions in the system to be implemented concisely (around 500 lines). In spite of this, the model checker uncovered several bugs that could deadlock the system. This highlights the limitations of careful code inspection and traditional testing, and the benefits of using tools like model checker that explore the various possible scheduling scenarios systematically.

The first bug was due to a circular dependency involving 3 processes that resulted in a deadlock. Once identified, the deadlock was avoided by eliminating the cycle.

The second bug involved a situation when the sliding window in the retransmission protocol was full and, therefore, not accepting any new messages to be sent to the network. This eventually led to no new data packets being accepted from the network. Since incoming messages were delivered in FIFO order, an explicit acknowledgement message that could unlock the system was trapped behind a data packet resulting in a deadlock. To fix this problem, packets have to be dropped occasionally to allow the explicit acknowledgements to get through.

Two other bugs uncovered were similar to the bug that was discussed in the example in Section 4.4. They would result in deadlocks if applications requested zero-byte data transfers.

The remaining bugs discovered involved receiving unexpected messages or not receiving expected messages. The first bug involved receiving acknowledgments with invalid acknowledgement numbers. This was fixed by first checking for the validity of the acknowledgement numbers before using them. The second bug involved receiving an unexpected import reply message. These messages are usually received in response to an import request. An unexpected reply would deadlock the system. The problem was fixed by adding code that discarded these unexpected messages. The final bug involved not receiving a reply to an import request. We had been aware of this bug but had not fixed it yet. This bug can be eliminated using a timeout.

5.2 Discussion and Limitations

The model checker is very effective in finding bugs. Spin was used to develop and debug the VMMC firmware implementation in ESP before running it on the device. So far, we have not encountered any bugs that were not caught by Spin. This is in contrast with our earlier firmware implementation in C where we encountered new bugs every time we tried a different class of applications or ran on a bigger cluster.

Model checking allows bugs to be uncovered early in the debugging process. This is highlighted by the fact that several bugs found by Spin would not have been discovered using conventional testing as long as our VMMC implementation was used on all the machines in the network. These bugs could only be triggered when the firmware was used to communicate with other VMMC implementations that generated unexpected messages because they were either malicious or buggy.

Partial searches are very effective in finding bugs in the concurrent programs. This is because the state machine being explored is usually much larger than necessary in which each state of the minimal state machine is represented multiple times. Techniques like abstraction and optimizations like partial-order reduction [20] try to eliminate some of this redundancy. However, significant redundancy remains because the size of the state space is exponential in the size of the model. For instance, a variable `i` whose value ranges from one to ten but has no bearing on the property can result in each state of the minimal state machine being explored ten times. So even a partial search that explores a small fraction of the state space can cover a significant fraction of the minimal state machine.

The effectiveness of using abstract models to check larger

[4] The Spin model checker was run overnight on a shared machine for over 12 hours. 3 hours and 57 minutes was the processor time that the model checker was allocated during this period.

[5] The firmware was used to run a number of microbenchmarks and applications on a 16 processor (4x4) cluster [24].

systems is demonstrated by the fact it could be used to find bugs in the VMMC firmware. Unfortunately, it is difficult to quantify the actual reduction in the size of the state space that resulted from abstraction or the fraction of the state space that was explored by the partial search. This is because the actual size of the state space can only be determined by exploring the entire state space. It is possible to compute an upper bound on the size of the state space.[6] However, this grossly overestimates the size of the state space because the state space tends to be very sparse.

One of the limitations of model checking is that it requires test code to be provided for each property to be verified. The test code is responsible for simulating the environment (external events like network message arrivals) as well as specifies the property to be verified. A mistake in the test code can result in the wrong property being checked or, more commonly, failing to explore a part of the state space for bugs. The latter happens when the environment is over constrained. For instance, when debugging the retransmission protocol, our initial code simulated a well-behaved environment that only generated network packets with the expected sequence numbers. Later, we found bugs when the test code was modified to generate unexpected messages. Another problem that we encountered when looking for deadlocks was that a deadlock that involved only a few processes in the model could go undetected. This is because a deadlock in Spin is a state out of which there are no transitions. When a deadlock involves only a few processes, the remaining processes can sometimes make progress. Consequently, there are always transitions out of the current state. Therefore, the state machine is not technically deadlocked. For instance, this happens when a deadlock in one part of the system prevents messages from being sent while another part of the system responsible for receiving messages from the network continues to function correctly. To avoid this, the test code can be changed to ensure that no part can inactive for long periods of time (by maintaining activity counters).

6 Related Work

6.1 Model Extraction Approaches

Model extraction by hand. Several researchers have verified various aspects of Operating Systems using model checkers. These efforts involved extracting an abstract model of the system by hand. Spin was used to verify the Interprocess Communication Subsystem in Harmony [7] (a real time operating system) and RUBIS microkernel [15]. The latter study found that significant effort was needed in extracting the model. Spin was also used to develop and verify a synchronization protocol for Plan 9 [27]. More recently, Spin was used to verify the IPC system of the Fluke OS [30]. All these studies found that the model checker was able to find some hard to find race conditions.

[6] If n is the number of bits needed to encode a state, then 2^n is an upper bound.

Automatic Model Extraction. To avoid the problems with model extraction by hand, some researchers have extracted the models automatically from the source code. Teapot [8] is a domain-specific language for implementing software cache coherence. It extracts a model that can be used by the Murphi model checker [13]. Promela++ [3] is a language for implementing layered protocol. Its compiler generates model that can be used by the Spin model checker. Esterel [4] is a language for specifying synchronous reactive systems and is primarily used in hardware design. The Esterel programming environment includes verification tools like model checkers that can be used to test the programs. Esterel was used to implement a subset of the TCP protocol [6]. They showed that esterel could be used to generate efficient code. However, they did not report any experience with the verification tools.

In all these cases, the domain-specific language is used to encode the control structure of the program. The rest of the program (data handling) is handled using a different language (typically C). The compiler for these languages extracts a single model that reflects the control structure of the program.

Java PathFinder [18] translates Java programs into Spin specifications. It handles significant subset of the Java including dynamic object allocation, object reference, exception processing and inheritance. However, they do not handle features like method overriding and overloading. Also, they do not provide a way to abstract details so that a tractable model can be extracted.

Verisoft [17] uses a different approach to perform model checking on a concurrent system. Instead of trying to extract a model, it explores the state-space of the system by replacing the scheduler of the concurrent system. By controlling the scheduler, it can force the concurrent program to execute all possible interleavings. This allows it to apply model checking to actual programs written in traditional languages like C (instead of a model). The problem is that it can explore much smaller state spaces because some of the techniques used by model checkers like Spin to optimize the exploration cannot be applied.

Automatic model extraction with support for abstraction. More recent efforts have focused on extracting several abstract models to verify different properties in the system.

FeaVer [22] extracts Spin models from programs written in a C dialect that has simple extensions to support event-driven state machines. The system allows the programmer to specify pairs of C and Spin code patterns. When the C pattern is encountered during translation, the corresponding Spin code is generated. This approach automates the extraction of abstract models. However, the translator does not have any semantic information to check the validity of the translation. The system was used to debug the call processing software for Lucent's Pathstar access server.

Lie et. al. [26] use an approach similar to FeaVer [22] to extract Murphi [13] models from C programs. It requires the programmer to specify two things: a set of patterns that iden-

tify the C code that has to be captured in the extracted model, and transformations that translate the identified C code into Murphi code. Unlike FeaVer, it uses program slicing [31, 29] to extract additional code that affects the identified code. However, the standard slicing algorithms have problems with C constructs like pointers, unions and unstructured control flow. Like FeaVer, it cannot check the validity of the generated model.

Bandera [11] allows automatic extraction of finite state models from Java programs. It uses techniques like program slicing [31, 29] and data abstraction to allow more tractable models to be extracted. ESP and Bandera differ in how nondeterminism is used during model extraction. In Bandera, nondeterminism is used only when an *undeterminable* value flows into a test of a conditional statement. In ESP, nondeterminism is used to assign values to all variables and fields that are not dropped by the abstraction but whose values become *undeterminable*. Another difference between ESP and Bandera is that the ESP language was designed to permit model checking. In contrast, Bandera targets Java that has a number of language features that are difficult to translate efficiently into models. So far, Bandera has been used to verify properties in only simple programs.

The SLAM project [1, 2] extracts a predicate abstraction to check assertions in sequential programs written in C. A predicate abstraction is a model with only boolean variables that correspond to conditions in the original program. The assertion is checked in the predicate abstraction using a model checker. Since the checker may generate false positives, symbolic execution is used to verify the counter examples generated by the model checker. If the counter example is invalid, the predicate abstraction is refined to eliminate the counter example. This approach has not yet been extended to handle concurrent programs.

6.2 Debugging System Software

A vast amount of research has focused on the problem of debugging system software. The techniques used span language design, model checking, compiler analysis, and runtime methods. In this section, we discuss some of the related work in this area.

As described in Section 6.1, model checkers have been used to debug system software. Some [7, 15, 27, 30, 18, 17, 22, 11] have focused on debugging programs written in general purpose languages like C, C++ and Java. Others have proposed domain-specific languages [8, 3, 4] that have been designed with model checking in mind, and therefore, allow model checking to be more effective.

Meta-level Compilation [9, 16] provides a framework for extending a compiler with application-specific code that can be used to statically check certain properties of that application. It was used to look for bugs in several systems including the cache coherence protocols for the FLASH multiprocessor and the Linux kernel. This technique requires little change to the source code and was very effective in finding several hundred bugs in these systems. The compiler extensions look for violations of properties like proper buffer allocation and deallocation, and absence of deadlocks. However, these extensions perform only intra-procedural analysis. In some instances, a separate global pass was used to combine data gathered by the intra-procedural analysis of the different functions to check a global property. Since the static analysis is inexact, it can generate false-positives. The bugs reported have to be double-checked by the programmer. In addition, the limited scope of intraprocedural analysis can generate false-negatives.

Eraser [28] detects data races in multithreaded programs. It instruments the program binary to check at runtime that a lock protects each shared variable access. It does not impose any constraints on the programs, and therefore works on existing programs with little modifications. However, the tool can detect only the data races that occur during the debugging runs; it is the programmer's responsibility to ensure that the program is run with several different inputs so that it is tested thoroughly. In addition, the instrumentation results in a factor of 10 to 30 slowdown in program execution. This can prevent some data races in the program from occurring during debugging.

Programming language features can often prevent an entire class of bugs. Safe programming languages prevent a program from accessing a dynamically allocated object after it has been freed. The Vault [12] language uses an expressive type system to enforce high-level protocols in system software. The type system allows a module writer to specify properties like "a read system call to read from a file can be called only after that file has been opened using the open system call".

7 Conclusions

ESP allows abstract models to be extracted from the programs. These models are smaller because they omit details that are irrelevant to the property being checked. In ESP, the programmer specifies the abstraction and therefore has control over the abstraction process. The ESP compiler uses the abstractions specified by the programmer conservatively when generating an abstract model. This ensures a bug in the ESP program will be in the generated model even when a programmer makes a mistake in specifying the abstraction.

Abstraction was essential for obtaining models that could be used to check for system-wide properties like absence of deadlocks. The earlier version of the compiler [25] was unable to find deadlock bugs in the VMMC firmware. The new version of the ESP compiler generated an abstract model that was successfully used to uncover seven deadlock bugs. Even with these models, only a partial search was possible. In spite of this, we have not encountered any new bugs while running the firmware on the device.

The use of model checker is greatly simplified when the models could be automatically extracted from the ESP programs by the compiler. Other studies [22, 11] have shown

that automatic extraction is possible even for traditional languages like C and Java. This not only increases our confidence that the model accurately reflects the program but also allows the system to be rechecked with little effort whenever changes are made to it.

A model checker is very effective as a debugging tool. Sometimes, only a partial state-space exploration is possible due to resource constraints. However, this is acceptable because the goal is to identify bugs and not to certify correctness. Even a partial systematic search by the model checker results in more extensive testing than traditional testing methods and can be invaluable in debugging concurrent firmware.

Acknowledgments

This work was supported in part by the National Science Foundation (CDA-9624099,EIA-9975011,ANI-9906704,EIA-9975011), the Department of Energy (DE-FC02-99ER25387), California Institute of Technology (PC-159775, PC-228905), Sandia National Lab (AO-5098.A06), Lawrence Livermore Laboratory (B347877), Intel Research Council, and the Intel Technology 2000 equipment grant.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Programming Languages Design and Implementation*, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *International Spin Workshop*, 2000.

[3] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A Language for Correct and Efficient Protocol Construction. In *Infocom*, 1998.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[5] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *International Symposium on Computer Architecture*, 1994.

[6] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *SIGCOMM*, 1996.

[7] T. Cattel. Modeling and Verification of a Multiprocessor Realtime OS Kernel. In *International Conference on Formal Description Techniques*, 1994.

[8] S. Chandra, B. E. Richards, and J. R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Programming Languages Design and Implementation*, 1996.

[9] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using Meta-level Compilation to Check FLASH Protocol Code. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[10] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically Closing Open Reactive Programs. In *Programming Languages Design and Implementation*, 1998.

[11] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, 2000.

[12] R. DeLine and M. Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In *Programming Languages Design and Implementation*, 2001.

[13] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[14] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects*, 1997.

[15] G. Duval and J. Julliand. Modeling and verification of the RUBIS μ-Kernel with Spin. In *International Spin Workshop*, 1995.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Operating Systems Design and Implementation*, 2000.

[17] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Principles of Programming Languages*, 1997.

[18] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. In *International Journal on Software Tools for Technology Transfer*, 1999.

[19] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[20] G. Holzmann and D. Peled. An Improvement in Formal Verification. In *International Conference on Formal Description Techniques*, 1994.

[21] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[22] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *International Conference on Software Engineering*, 1999.

[23] S. Kumar and K. Li. Dynamic Memory Management for Programmable Devices. In *International Symposium of Memory Management*, 2002.

[24] S. Kumar and K. Li. Performance Impact of Using ESP to Implement VMMC Firmware. In *Workshop on Novel Uses of System Area Networks (SAN-1)*, 2002.

[25] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A Language for Programmable Devices. In *Programming Languages Design and Implementation*, 2001.

[26] D. Lie, A. Chou, D. Engler, and D. Dill. A Simple Method for Extracting Models from Protocol Code. In *International Symposium on Computer Architecture*, 2001.

[27] R. Pike, D. Pressoto, K. Thompson, and G. Holzmann. Process sleep and wakeup on shared-memory multiprocessors. In *EurOpen Conference*, 1991.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *Transactions on Computer Systems*, 15(4):391–411, 1997.

[29] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[30] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Hot Topics in Operating Systems*, 1997.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.