Formal Methods: A Practical Tool for OS Implementors

Patrick Tullmann Jeff Turner† John McCorquodale Jay Lepreau Ajay Chitturi Godmar Back

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

{tullmann,sjt,mcq,lepreau,chitturi,gback}@cs.utah.edu
http://www.cs.utah.edu/projects/flux/

Abstract

The formal methods community has long known about the need to formally analyze concurrent software, but the OS community has been slow to adopt such methods. The foremost reasons for this are the cultural and knowledge gaps between formalists and OS hackers, fostered by three beliefs: inaccessibility of the tools, the disabling gap between the validated model and actual implementation, and the intractable size of operating systems. In this paper, we show these beliefs to be untrue for appropriately structured operating systems.

We applied formal methods to verify properties of the implementation of the Fluke microkernel's IPC subsystem, a major component of the kernel. In particular, we have verified, in many scenarios, certain liveness properties and lack of deadlock, with results that apply to both SMP and uniprocessor environments. The SPIN model checker provided an exhaustive concurrency analysis of the IPC subsystem, unattainable through traditional OS testing methods. SPIN is easily accessible to programmers inexperienced with formal methods. We present our results as a starting point for a more comprehensive inclusion of formal methods in practical OS development.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

† U.S. Department of Defense.

1 Introduction

Operating system implementations are at once concurrent, non-deterministic, and usually optimized for speed over safety. Nevertheless, we demand a high degree of reliability from them. Formal design and analysis were developed for exactly these conditions [6, 10, 12]. Although hardware designers are successfully adopting formal methods into their development cycles [20], most OS programmers are probably not aware of these methods due to the gaps between cultures and publication venues. We believe that OS programmers who are aware of these methods maintain three debilitating perceptions of formal methods: a steep learning curve, a distance between the validated model and the actual implementation which invalidates any conclusions, and that operating systems are too large for these methods to apply. Based on the experiment reported in this paper, we believe that in many instances these are *mis*perceptions.

Before we proceed into the discussion of formal methods themselves, we provide a short description of the motivating example—the Fluke kernel's IPC subsystem [8, 9]. Then we discuss similar efforts to bring verification and operating systems together. Section 5 describes the method we used to validate Fluke's IPC implementation. Then we present some specific results, some open issues, and our conclusions.

2 Interprocess communication in Fluke

Fluke [8] is a "small" microkernel in the vein of L4 [13], providing interprocess communication (IPC), memory mapping (but not virtual memory), and synchronization primitives; it was designed to run on both uniprocessors and multiprocessors. Fluke had been running for about six months, though only on uniprocessors, when this modeling effort began. Our project centered around Fluke's reliable IPC facility, about 3000 "lines" (a non-comment line of code that contains more than just punctuation symbols) of C code—approximately one quarter of the core kernel. Fluke's support for sophisticated resource management and relatively efficient systems is dependent on the reliability and performance of its IPC facilities [9], which has a number of interesting and subtle aspects.

A reliable IPC link established between client and server threads in Fluke is a connection-based, reversible, half-duplex stream. The server is defined as the thread that blocks and waits for a connection, while the client is defined as the thread that connects with a blocked server thread. Reliable IPC is implemented as a non-trivial protocol involving multiple fine-grained locks on each participant, and multiple wait queues. Client and server directly manipulate each other's state, and may block waiting for the other. Additionally, as a fully reliable protocol, both sides of the connection must remain consistent.

The factor which most complicates the implementation is the interruptibility of Fluke's reliable IPC [19]. Internally, threads can post *cancel* operations to other threads. When the target thread handles a cancel it must cleanly backtrack, undoing part or all of the system call it is performing; analogous to a processor's handling of precise exceptions.

In sum, Fluke IPC is important and highly concurrent with a complex implementation, making it a worthy target of formal methods.

3 Formal methods

Formal methods attempt to insure the quality of complex systems by bringing the strong, consistent world of mathematical logic into the process. Such methods can establish a much more general class of properties of all the executions of the system model than testing methods, which typically establish simple properties of a small subset of the possible execution scenarios.

Currently, there are two major classes of tools used in the formal verification of a hardware or software system: the *theorem prover* and the *model checker*. Theorem proving begins with a set of axioms, preconditions, and other knowledge of the system. Using this knowledge, hypotheses about the system are proven. Assurance in the system is gained from proven hypotheses which state desirable invariants of the system. Theorem proving based verification is capable of dealing with generalized (often infinite-state) models of the system and can prove properties involving data as well as control flows.

In contrast, model checkers exhaustively explore the possibilities of a system model. Given a model of concurrent threads, a model checker will explore all of the interleavings of those threads reachable from the initial setup—the state space of the model. The model checker indicates a deadlock if it reaches a situation in which none of the threads can progress. Livelock situations—progressing without accomplishing anything—are reported if the model checker can detect cycles in its explorations.

The reachable state space may grow very quickly, and can easily make model verification impossible because of resource exhaustion—typically main memory. The key to making this problem tractable, yet still preserving useful verification properties, is to *abstract* secondary details out of the model. For example, when checking for deadlocks in a communication synchronization protocol, the actual data communicated is of secondary importance. Simply maintaining an indication that data was sent is sufficient; the actual content of the data is not relevant.

State of the art theorem provers, for example the popular PVS system[15], still require substantial skilled human intervention to guide the prover through nearly all non-trivial proofs. By contrast, our experience demonstrates that model checking is accessible to relatively naive users. Additionally, model checkers provide more effective reports when errors are encountered—generally a full trace from the beginning of execution. In our experience the expression of a model (in a special modeling language) and the expression of a program (in a standard programming language) can be very close. On the other hand, the resource scaling problem encountered by model checkers is not present in theorem proving.

Formal methods research has made major advances in the last decade. In particular, advances in model checking, especially with the state space explosion problem, have made it a practical tool; it is finding increasing use in the development of complex hardware systems, for example the UltraSPARC processor[20]. It is our belief that the same results and success can be found by applying formal methods to large scale software systems—in particular operating systems.

SPIN

In SPIN [11], a popular model checker developed at Bell Labs, formal models are written in PROMELA which is loosely based on Dijkstra's guarded command language notation [6] and Hoare's CSP [10]. For this paper the most important facets of PROMELA are that it looks and acts much like a standard procedural programming language (although an impoverished one), and that it implicitly supports blocking a "thread."

SPIN does incorporate special features to attack the state-explosion problem. Partial-order reduction [16] reduces interleaving by grouping consecutive statements that have no global effects. It fully preserves accuracy while reducing the state space and run time. Bit state hashing [5], or "supertracing," dramatically reduces the memory requirements of model verification by bit hashing the states without collision detection; unlike partial-order reductions, this technique may miss on the order of 1% of the total state space.

4 Other operating system verification efforts

We are aware of two previous efforts to verify kernel subsystems with SPIN, for Harmony [3], a "portable real-time multitasking multiprocessor operating system"; and for the RUBIS microkernel [7]. Although the modeled subsystems were less complex than ours, both of these results bolster our position that formal methods are worth applying to operating systems. Additionally, considering the many recent improvements made to SPIN the conclusions drawn by the former might be stronger today. However, they were published in formal methods venues obscure to the mainstream OS community, limiting their impact. In a more tightly focused effort, the Plan9 developers successfully used SPIN to debug a synchronization protocol [17].

A great deal of work has been done on the specification and analysis of the Mach kernel. Computational Logic Inc. developed a mathematical model of the Mach 3.0 microkernel [2]. This work focuses on a model of the Mach API, as opposed to our focus on implementation details. CLI and OSF [14] are also using the Mach specifications to generate conformance tests for the MK++ microkernel. The specifications are used to identify legal state predicates which can be inserted into the kernel as assertions. This project is similar to ours in its focus on making formal methods practical for software engineers.

Unity [12] is a concurrency formalism used to specify and verify concurrent programs. It was used to specify requirements for the resource allocation portion of Bull's GCOS I/O subsystem and to develop and verify an abstract algorithm based on the specifications [18]. This algorithm was used as the basis for the implementation of the I/O subsystem. Our work differs primarily in our implementation-driven approach—the tight coupling of the PROMELA code to the kernel source code.

5 Our approach and methodology

Since we were verifying an existing system, we wanted the model to be tightly bound to the implementation. For this project, we were not interested in a high-level model of the design specification, as errors in such a model would not easily map back to the implementation. Since none of the verifiers knew the kernel IPC system, we needed a simple method for generating the model. We also needed to limit the size of the model wherever possible, to reduce the size of its state space, while preserving relevant verification properties.

```
/*
 * ipc_reliable_transfer(sender, receiver)
 */
assert(sender == currentThread);

if
:: TRUE ->
     /* Success */
     threads[receiver].recv_payload =
           threads[sender].send_payload;
     /* RETURN */ rc = 0;
:: TRUE ->
     /* Receiver page faulted */
     threads[receiver].wait.resume_rc =
           KR_PAGE_FAULT;
     thread_ready(receiver);
     /* RETURN */ rc = KR_RESTART;
:: TRUE ->
     /* Sender page faulted. */
     thread_ready(receiver);
     /* RETURN */ rc = KR_PAGE_FAULT;
fi
```

Figure 1. The PROMELA implementation of the complex `ipc_reliable_transfer()` function. The `if` is PROMELA's non-deterministic switch statement.

The structure and design of the kernel were a key aid to our effort. Since Fluke is written entirely in well structured C, all data abstractions are well maintained. Additionally, Fluke (coincidentally) uses a well-defined subset of C: no variable argument functions, no recursion, and all functions returning one of five error codes. These properties made the model generation fairly simple. Importantly, the kernel wasn't designed with verification in mind, so no sacrifices were made on the behalf of this effort.

For a given section of code, we had two choices. We could either *translate* it into PROMELA—a simple (but not automated) transformation. Or, we could *abstract* the details away and implement in PROMELA "from scratch." There are two reasons to re-implement from scratch. First, if the section of code implements functionality that PROMELA inherently supports, it makes sense to take advantage of that support, if it does not detract from the focus of the verification. For example, a mutex implementation in C must keep track of blocked processes and mutex state, in addition to the synchronization of accesses. PROMELA supports mutexes trivially via guard expressions. In order to take advantage of these features we re-implemented the Fluke mutex abstraction in PROMELA from scratch. The other reason to abstract is to avoid modeling unnecessary detail. For example, the `ipc_reliable_transfer()` routine must deal with page fragments, permissions, and scatter/gather buffers; it returns success if the data was copied, or an error indicating that the source or the destination page-faulted. For our

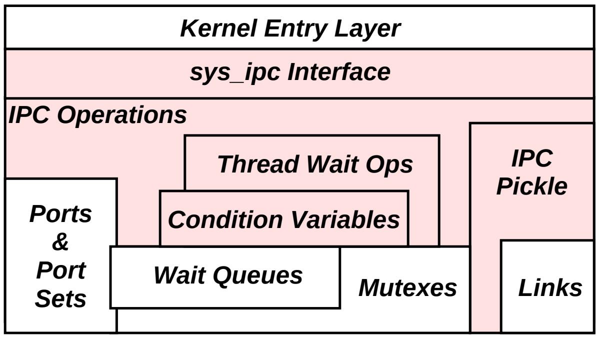

Figure 2. The breakdown of translation and abstraction of different areas of the system. White areas were *abstracted*: re-implemented in PROMELA independent of the C source. Shaded areas were *translated*: a mostly line-for-line translation of the C source into PROMELA. Note that the relative sizes of the areas mean nothing.

model, all that is important is whether one of the IPC participants faulted. The exact nature of the fault is irrelevant to the core IPC code. The new `ipc_reliable_transfer()` procedure is shown in Figure 1.

Figure 2 shows which parts of the IPC system were translated, and which were abstracted. Because the implementation of each section is independent, they can each be translated or abstracted independently. It turns out that abstracted portions constituted only one fourth of the total modeled code. More importantly, the abstraction was primarily limited to core kernel primitives (the lower layers in the figure), meaning that all of the complex logic in the IPC system was translated, further increasing our confidence in the results.

6 Verification scenarios and results

We found the resources offered by a low cost machine to be adequate for our tests. A 200MHz Pentium Pro with 256 MB of memory was used throughout, taking at most 90 minutes to complete a verification, and generally about 10 minutes. We also found low cost human resources to be adequate, with four OS graduate students new to formal methods spending a total of five man months on the modeling and testing effort.

All of the scenarios, or verification models, we have currently verified have the same structure. They are written just as user-level processes are written for the actual system. They can make the same system calls, and expect the same return values. Most of verification models consist of one "user-level thread" acting as the server, and another acting as the client. Because SPIN explores *all* interleavings of the system, the results of a verification apply to the kernel running on an SMP system, and to a preemptive kernel running on a uniprocessor.

For example, in a very simple model the client thread will repeatedly connect to the server, send a byte of data, and disconnect. The server thread will repeatedly wait for a connection, receive a byte of data, and disconnect from the client. Each 'send a byte of data' operation can potentially fail with a page fault on either the client or server side. SPIN will fully explore the state space of this scenario, reaching every state from the initial condition. SPIN's completion indicates that there are no deadlocks or livelocks in this scenario.

At this time we have searched for livelock in only a few models. The first time we ran the livelock test, SPIN reported a single failure—an infinite series of page faults.

Our growing collection of verification models has uncovered two serious bugs, and increased our confidence in Fluke's IPC path—though quantifying this is difficult. We consider the complexity of the two bugs uncovered as a good argument for the comprehensiveness of SPIN's analysis and our verification scenarios.

The first bug involves a missing call to `mutex_unlock()`; it was discovered in a verification scenario consisting of a client, a server, and a cancel thread issuing random cancels. In the current implementation of Fluke IPC, only a single process will ever be executing in certain areas of the kernel, while on an SMP machine or with a preemptive version of the kernel, several processes can be in these areas. SPIN discovered a scenario where, in one of these areas, if a cancel had been flagged, a mutex was not unlocked when it should have been. SPIN presented a complete trace of the scenario leading up to the resulting deadlock. Because of the close coupling between our model and the implementation, no effort was required to find and fix this bug in the C code; the bug was in the "same place" as in the PROMELA code.

The second bug we have discovered, so far, is a race condition in the thread cancel handler. If a first thread was in the process of handling a cancel, and a second thread set the cancel pending flag of the first, this second cancel pending could be ignored.

We are still developing verification models, and are looking at generating a single, large model that will explore all interleavings of a client and server through all IPC system calls. In addition to deadlock and livelock, we want to use SPIN to test that the correct user error codes are returned in all situations.

7 Issues

Although we have demonstrated that formal methods, in particular the SPIN model checker, can be a practical tool for

OS implementors, there remain significant questions concerning such methods.

One question is the *scale* of the software to be validated. State space explosion limits the size of the subsystem that can be examined. Modeling our IPC system was tractable, but modeling the entire kernel would not have been. Another issue is the *composition* of properties from separately modeled subsystems, which is generally infeasible with a model checker because the interleavings between the subsystems can get lost. Developments in composition would greatly aid, and in some cases enable, formal verification of very large systems. A third issue is *maintenance*: as a system's implementation evolves, how can the model track it, if both model and implementation exist separately. A fourth issue is *cost effectiveness*: how long would it have taken to find these bugs through normal testing? We believe the first one (the missing unlock) could have been found quickly but cannot say for the second. On the other hand, we were able to quickly code and test multiple different "fixes" to the race condition. This is certainly not possible with traditional methods. Because the Fluke implementation hasn't been debugged on an SMP machine we cannot accurately compare the costs of traditional debugging with using a model checker for certain classes of bugs. In any case, we feel we are now assured that a large class of concurrency bugs are not present in Fluke.

Additionally, there are issues related to the approach we developed to applying formal methods to an operating system.

First, while we were able to "translate" much of our C code into PROMELA, it is not clear how much of this process could be automated. The decision about what to translate and what to abstract still needs to be made. Perhaps another language could generate both the model and the implementation, as was done in the Teapot work [4] on software cache coherence, or very recently with Promela++ [1]. This would also solve the maintenance problem mentioned above.

Second, the structure of the kernel's implementation was helpful in generating the PROMELA models, just as this structure would help if the kernel were ported to a new processor architecture. It would be useful to codify the subset of C used and discover which constructs help or hinder the translation/abstraction process.

8 Conclusion

Our experiment demonstrated that the SPIN model checker required only a short learning curve, that complex kernel subsystems were tractable to model, and that subtle bugs could be found that might have evaded traditional testing techniques. Since we directly derived the model from the kernel source code, instead of from a specification, we have high confidence that our results apply to the actual implementation. That the kernel is written in a stylized manner, in a restricted subset of C, made nearly direct translation possible for the majority of the modeled code. A simple implementation language is a good match for the modeling language. That the kernel is a microkernel means that a model of a significant portion of it could be verified.

We believe that formal methods, and model checking in particular, should be pursued more aggressively as a component of today's practical OS development. Research into the hard issues of applying model checking to OS development should be pursued as well.

Availability

Our complete PROMELA sources, some simple tools, slides from several presentations, and project reports are available off of *http://www.cs.utah.edu/projects/flux/*. More information on Fluke, including its API and several related papers are also available from that page. A source distribution of Fluke will be made in the coming year.

Acknowledgements

The authors thank the Fluke development team for providing something to model, especially Bryan Ford who wrote the majority of the IPC code. We thank the entire Flux group for varied support and feedback. The anonymous reviewers provided helpful and encouraging comments, and we especially thank Ganesh Gopalakrishnan who provided the impetus and initial direction for this project.

References

[1] A. Basu, M. Hayden, G. Morrisett, and T. v. Eicken. A language-based approach to protocol construction. In *Proc. of the ACM SIGPLAN Workshop on Domain Specific Languages (WDSL)*, Paris, France, Jan. 1997.

[2] B. Bevier and L. Smith. A mathematical model of the Mach kernel. Computational Logic Technical Report CLI Technical Report 102, Computational Logic, Inc., Sept. 1994.

[3] T. Cattel. Modelling and verification of a multiprocessor realtime OS kernel. In *Proceedings of the Seventh International Conference on Formal Description Techniques*, Oct. 1994.

[4] S. Chandra, B. Richards, and J. Larus. Teapot: Language support for writing memory coherence protocols. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, May 1996.

[5] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[6] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of progams. *CACM*, 18(8):453–457, Aug. 1975.

[7] G. Duval and J. Julliand. Modeling and verification of the RUBIS μ-Kernel with SPIN. In *Proceedings of the First SPIN Workshop*, Oct. 1995.

[8] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available under http://-www.cs.utah.edu/projects/flux/fluke/html/, 1996.

[9] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[10] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, Aug. 1978.

[11] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[12] J. M. K. Mani Chandi. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

[13] J. Liedtke. On micro-kernel construction. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[14] Open Group Research Institute and Computational Logic, Inc. OSF formal methods project. http://www.osf.org/-formalmethods/, 1996.

[15] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga Springs, NY, June 1992. Springer-Verlag.

[16] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in Systems Design*, 8:39–64, 1996.

[17] R. Pike, D. Pressoto, K. Thompson, and G. Holzmann. Process sleep and wakeup on shared-memory multiprocessors. In *Proc. of the Spring 1991 EurOpen Conference*, pages 161–166, May 1991.

[18] M. G. Staskauskas. Formal derivation of concurrent programs: An example from industry. *IEEE Transactions on Software Engineering*, SE-19(5):503–528, May 1993.

[19] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 85–88, Seattle, WA, Oct. 1996. IEEE.

[20] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Lowenstein. System design methodology of UltraSPARC-1. In *Proceedings of the Thirty-Second Design Automation Conference*, pages 7–12, San Francisco, CA, June 1995.