Fluke IPC Verification Project Report

Patrick Tullmann Godmar Back Ajay Chitturi
John McCorquodale Jeff Turner

`veripc@jensen.cs.utah.edu`

November 27, 1996

Abstract

We have formally described and modeled the reliable Interprocess Communication (IPC) as implemented in the Fluke operating system[4]. With this model we were able to verify certain properties, such as freedom of deadlock, under a number of test scenarios. We have also provided a base of documentation and infrastructure for formal verification of other areas of Fluke. Our model tries to follow closely the actual C implementation of the IPC system, allowing us to make strong claims about the actual system, as opposed to claims simply about the design.

Contents

1 Introduction

Fluke is a microkernel implemented in an object-based spirit. Fluke operations are made available as methods on a collection of primitive *Fluke objects* such as threads, ports, memory regions, mutexes, condition variables, and others. Fluke's high-level organizing pattern is that of *recursive virtual machines* or *nested processes*[4]. Nested processes communicate via the low-level Fluke inter-thread communications facility, called *Fluke IPC*, to provide a virtual machine abstraction.

Fluke allows the kernel state of an object to be exported *at any time* [7]. This even holds for long-running kernel operations in which an object may be involved. Therefore, these operations must be decomposable into smaller units that leave each object in a consistent state which can be exported. A checkpointer may extract an object's state, pickle it and use the exported state to reconstruct and restart these objects at a later time.

For instance, the Fluke IPC mechanism is designed to allow suspension and resumption of long-lived IPC operations. The operations are broken into smaller pieces. Objects enter and leave these smaller primitives in a consistent state. Since the Fluke thread objects of the participating threads are used to keep track of the IPC operation's progress, these consistent states must be valid restart-points for these threads.

Our work thus far has focused on model-checking the Fluke IPC subsystem, in particular the Fluke *reliable IPC* mechanism, which is the most complex, subtle and important of the three flavors of Fluke IPC.

The achievements of this effort are threefold. First, we have reverse-engineered a descriptive model of the Fluke IPC subsystem from the current C implementation. Second, we used this descriptive model to develop a checkable model of the reliable IPC subsystem in Promela [5] which we present in sections 2 and 3. Lastly, we have begun evaluating the performance of this model under various realistic scenarios with encouraging results, as presented in section 4. This section we also describe our experiences with SPIN and list some perceived shortcomings. We conclude with an evaluation of the prospects of using the SPIN model-checker as the basis for a larger-scale formal verification effort of the Fluke kernel.

2 IPC and Threads

2.1 Overview of Fluke IPC

This section gives the basics of the Fluke reliable IPC mechanism. Two other forms of IPC in Fluke, One-way and Idempotent, are described in [3].

Reliable IPC is the mechanism Fluke provides for general, high-performance, reliable, protected communication. Reliable IPC in Fluke operates in a stream-oriented model, in which two threads connect to each other directly and can send data to each other as if through a pipe. These connections are symmetrical but half-duplex, meaning that data can only flow in one direction at a time. However, the direction of data flow can be reversed without destroying the connection.

Any Fluke thread can be involved in two reliable IPC connections at once: any thread may have both a connection to a *client* thread–a thread that "called" it, as well as a connection to a *server* thread–a thread that it has "called".

Either side of a reliable IPC connection may be held up arbitrarily long while waiting for the other side. Thus, there are no guarantees regarding the "promptness" with which a server thread can perform its work and disconnect. A server thread is bound to a particular client thread for the duration of the IPC operation.

Communication within the kernel is based on unilateral synchronization–interacting threads are required to cooperate with each other. Threads effectively "ask" other threads to complete an operation on their behalf, blocking until the result is available. For example, during a standard reliable IPC operation, the server thread will block waiting for a client to show up. It blocks in a state labeled as "waiting to receive". The client will grab the server and set up the connection. Once the connection is established, it changes the *server's* state to reflect an active connection (just in case they get disconnected.) and then starts transferring data to the server. When the transfer is completed, the client puts the server back on the ready queue so that it may run. When the server wakes up it will restart as if it had been doing a receive operation. The client will go to sleep in a state indicating that it has finished a send. The server wakes up and notices the client is in the right state, and then restarts the client, and returns to the user itself.

It is important to note that while the thread is blocked in a specific "wait state" it will not be restarted until its counterpart in the reliable IPC operation restarts it.

The IPC implementation uses *wait state values* to keep track of the state in which a particular thread is waiting. For example, if a thread is currently a server (in an IPC connection) and is waiting to send data, its wait state value will be `WAIT_IPC_SRV_SEND`. The set of wait states forms a tree whose leaves are single states. The leaves are represented in a binary encoding such that a single bit test suffices to determine whether a state belongs to a certain subtree, i.e. is a member of a subset of states. See Figure 1 for the tree of valid wait states.

```
WAIT_NONE
WAIT_ANYTHING ->
     WAIT_IPC_PICKLE
     WAIT_STOPPED
     WAIT_CANCELABLE ->
             WAIT_ON_COND
             WAIT_IPC_SRV_DSCN
             WAIT_IPC_SRV_RECEIVER ->
                     WAIT_IPC_SRV_RECV
                     WAIT_IPC_SRV_ASEND
             WAIT_IPC_SRV_SENDER ->
                     WAIT_IPC_SRV_ORECV
                     WAIT_IPC_SRV_SEND
             WAIT_IPC_CLI_DSCN
             WAIT_IPC_CLI_RECEIVER ->
                     WAIT_IPC_CLI_RECV
                     WAIT_IPC_CLI_ASEND
             WAIT_IPC_CLI_SENDER ->
                     WAIT_IPC_CLI_ORECV
                     WAIT_IPC_CLI_SEND
             WAIT_IPC_UNPICKLE
             WAIT_IPC_IDEMPOTENT ->
                     WAIT_IPC_EXCEPTION
                     WAIT_IPC_CALL
                     WAIT_IPC_SECURITY
```

Figure 1: Valid wait states for blocked threads

2.2 Promela Implementation of Fluke IPC

The Promela model of the IPC code resembles the implementation very closely. Very few details have been discarded or replaced with implementations of equivalent abstract semantics. The majority of functions which model the reliable IPC path are implemented by directly translating the corresponding C functions to Promela. (The C functions are described in some detail in the Appendix.) For example, the following is a snippet of C that implements the `sys_ipc_client_connect_send_over_receive()` function–the kernel interface for that function.

```
struct s_thread * const client = current_thread();
int rc;
```

```
rc = ipc_client_1_connect_send(client,
                               fluke_ipc_client_send_over_receive__entry);
if (rc) return rc;

return ipc_client_2_over_receive(client, WAIT_IPC_SRV_RECV);
```

Differing naming conventions aside, it can be seen that the "equivalent" Promela looks almost identical:

```
ipcClient1ConnectSend(currentThread,
                      threadExcConnectArg(currentThread),
                      ENTRYPOINT_CLIENT_SEND_OVER_RECEIVE);
if
:: rc==0 ->
     ipcClient2OverReceive(currentThread, WAIT_IPC_SRV_RECV);
     /* sets rc, RETURN rc */
:: else ->
     /* RETURN non-zero rc */ skip;
fi;
```

Since we don't model ports and port sets, the id of the server thread is passed as a parameter (this is accomplished by `threadExcConnectArg(currentThread)`).

2.2.1 Transfer of data

In the C implementation, the function `ipc_reliable_transfer()` copies the data and references from the sender space to the receiver space. This functions implements scatter/gather buffers, the bookkeeping needed for reference transfer, and other things – all of which are outside the scope of our verification.

In our Promela implementation, we simplify the data transfer to its essence. Only a single byte of payload (which is part of the thread's exception state) is transferred from the sender to the receiver.

The possibility of either the sender or the receiver page faulting during the transfer is implemented using the non-determinism that Promela provides. Note that the payload is copied only in the case in which there is no error. Additionally, because transfer is isolated to a single function we are able to make the inclusion of page fault a configurable option.

2.3 Threads in Fluke

Threads are a conventional execution context in Fluke. They run in an address space and access system services through a defined system call interface. The interface is object-based. Threads not only manipulate mutexes or other kernel objects through that interface, but also each other. For example, a thread may stop another thread simply by setting its state to `STOPPED`.

Threads in kernel space can directly access the state of other threads, provided they obey the locking protocol. Thus a client thread in an IPC operation can directly manipulate the state of the server thread when sending data to it. Another example is when a thread needs to cancel another thread – stopping its current execution and making it roll back to a clean entry point. First it locks the thread object, then it sets the target thread's "cancel pending" bit, and blocks. When the target thread undoes its current operation and enters the code to handle the cancel, it will notice the blocked thread and wake it up.

2.3.1 Thread State

There are four relevant sections of the Fluke thread state to our model, each of which is protected by an associated mutex. First is the generic state that all objects in Fluke have, this identifies the type of the object, whether it is active, etc. Second is the state associated with any outstanding IPC connections. Next is the state describing a blocked thread (the wait state). Last is the mis-named exception state which contains the state of the thread's user-kernel boundary.

2.3.2 Scheduling

Threads may be blocked, ready or running. At any time, at most n threads may be running, where n is the number of processors in the system. Unblocking a thread is accomplished by putting it on the ready queue. A dispatcher is then invoked which will eventually schedule that thread. The kernel is designed to run on SMP machines and have multiple threads executing concurrently.

2.4 Promela Implementation of Threads

Threads are modeled using Promela's *proctype* construct. Each thread is represented by a proctype which calls a particular function (implemented using a macro) corresponding to the kernel entrypoint. All thread state is stored in a global array of structures, one element per thread. The specific state modeled is described below.

Initially we experimented with using two separate proctypes to encapsulate the state and control portions of the thread. We expected this to hide a lot of information in local variables and thereby enable SPIN to optimize better. However, this approach involved a lot of communication between the state and control proctypes and so disabled extensive use of SPIN's *d_step* construct which allows *us* to optimize the control states. Since specifying liveness conditions using SPIN's never automaton requires access to global variables, we decided to take the approach of using one proctype per thread and made all thread state global.

2.4.1 Thread State

In our paradigm of translation of C into Promela, we kept much of the structure of the thread state in our models. Each thread is comprised of some 4 parts. There are is the wait queue information (next, and prev pointers); the IPC state information; information about the thread's current wait state; and the exception information.

The IPC state contains both explicit "pointers" to the active client and server threads, and models of pickled "links". It includes some flags of the current state of the IPC. The wait state includes a value indicating the current state the thread is in, if it is blocked, the condition variable it is blocked on (if it is blocked on one.) The cancel pending flag and the resume return code to use after unblocking. The somewhat mis-named exception state includes all of the data used for communicating with user mode–additionally it stores the "instruction pointer" reference used in rollback–See the Kernel Entry Layer section below.

The only state which does not have a direct correspondence from the C code to the Promela code is the rendezvous point and the data payload, both in the IPC section of the thread state.

2.4.2 Scheduling

We make use of SPIN's interleaving of the proctypes to model the thread scheduling. So by default any of the threads is runnable. If a thread has to *block* (for example while waiting on a condition variable), it sets a particular bit in its global state and waits for some other thread to reset this bit. The thread which resets the bit *unblocks* this thread.

Since SPIN tries all possible interleavings of the code during verification, we model an SMP scenario in which each of the runnable threads gets a processor to run. This also means that the results obtained by this approach do not depend on any scheduling algorithm. Modeling the single-processor case would actually be more difficult(!) since we would have to come up with a mechanism where by only one thread could run until it explicitly gave up control of the processor.

3 Fluke Core Models

While evolving our current Promela model of Fluke IPC we have developed a support infrastructure which includes Promela implementations of many low-level Fluke objects used during Fluke IPC sessions. We call this support infrastructure the *Core Models*. These models model such Fluke entities as mutexes, condition variables, wait queues, links, ports and port sets, and the Fluke kernel entry layer (which embodies behavior intimately related to the cancellability of IPC operations).

Implementation details and the design rationale of these models are presented here. Where appropriate, the implementation of these models is quite faithful to the C-language source from which they are derived. When particular divergence from the C-code has occurred, we note the underlying motivation and discuss the semantic similarities between the two implementations.

Since our model is heavily based on the actual implementation, we have a high measure of confidence that the results obtained from model checking are in fact applicable to the real system.

3.1 Translation Artifacts and the Sliding `d_step` Technique

Since Promela does not support language-level procedures or named blocks, we were forced to rely on C preprocessor macros to give us the ability to name code segments and provide a function-like decomposition of source code. Since these macros are not lexical scopes, and many instances of the same macro can be instantiated in a single lexical scope (proctype), these macros cannot contain variable declarations, labels or other symbol declarations or duplicate symbol declaration will occur. Thus, most macros assume the presence of a local variable called `rc` which is used to hold the "return value" from the "function" that the macro embodies. This variable can be interrogated by the user of the macro.

The Core Models are designed to allow the use of a technique we call the *sliding* `d_step`. When working with a model as large as the Fluke IPC model, the problem of state-space explosion is an omnipresent threat. Since the Core Models are largely self-contained and contain no external references, individual Core Models (or groups of them) can be verified together, and then their "public methods" can be encased with the SPIN/Promela `d_step` statement, which will avoid an overall increase in state-space. The positioning of these `d_steps` is quite flexible due to the hierarchical and rigidly encapsulated structure of our models. Thus, problems can be kept at the edge of manageability, to verify the Fluke implementation as rigorously as possible using the hardware available.

3.2 Mutex Model

Mutexes are a mechanism for achieving mutual exclusion among a set of concurrently executing threads. A mutex is an object that can be in one of two states, *locked* and *unlocked*. Mutex objects must support *lock* and *unlock*, which move the mutex from the unlocked state to the locked state, and vice versa. Mutual exclusion is guaranteed since the lock and unlock operations are atomic and only one contender will successfully acquire the lock. Attempts to lock a mutex that is already locked by some other thread will cause the calling thread to wait until the mutex is unlocked.

Fluke `mutex` Objects The Fluke implementation of mutexes uses Fluke wait queues to keep track of threads waiting to acquire a particular mutex.

The Promela `Mutex` Type Since Promela has builtin atomicity support via the `atomic` and `d_step` constructs, it was not necessary for us to build our model Mutexes on top of our model WaitQueues. Rather, one bit is sufficient to represent the state of a mutex. We call Mutexes with one state bit *simple* mutexes. They are implemented in `src/fluke/Mutex-simple.pr`.

However, there are many useful properties that one may wish to verify about mutexes and how they are used by other code. For example, it may be desirable to assert that the same thread that locked a mutex must unlock it. Also, we might wish to ensure that the *same* thread does not try to lock a mutex it already holds since this action would result in deadlock. We have a second, more heavyweight, implementation of Promela Mutexes, called *safe* mutexes that address these concerns and aggressively make sanity assertions. This implementation can be found in `src/fluke/Mutex-safe.pr`.

Both implementations of Promela Mutexes support the same set of functions. The intention is that `Mutex-safe.pr` will be used to verify the appropriate use of Mutexes in a system, and then `Mutex-simple.pr` will be substituted when the whole system is tested, to minimize state vector length.

3.3 Wait Queue Model

A wait queue is a data structure used to represent a FIFO list of threads waiting for a particular event. This is used to keep track of the list of threads waiting on a condition variable.

Wait Queues are implemented as a doubly linked list of threads where the Thread id are used as pointers. Each thread object contains one "next" and one "prev" pointer – this is sufficient since a thread can only be on one wait queue at a time. The interface provided by our model is exactly the same as that provided by the kernel's C implementation. In addition, the Promela implementation allows to remove an arbitrary thread from the wait queue.

3.4 Condition Variable Model

The C implementation of condition variables makes use of POSIX-like condition variables. In addition the interface provides a call which allows a thread to place another (captured) thread asleep on a condition variable. Unlike POSIX conditions variables, the associated mutex is not re-acquired before returning from a wait operation.

A condition variable is modeled using a **wait queue** of the threads waiting on the condition variable. There is no extra state required to be maintained as the wait queues implement all the required features.

3.5 Link Model

Existing, usable IPC connections can be in two states: active, and pickled. Active connections are internally represented as a direct pointer from on thread to another. (ie, `client->ipc_state.server` is the address of the server's s_thread_t object). Pickled connections are represented as *links*. One or the other is accurate, never both (the non-active link is 0.) A direct pointer can be thought of as a 'cached' value of the link.

The kernel provides **links** which represent inter-object references. They are used for encapsulating pointers and for reference counting. Initially we will model these as simply as possible; we may add verification of the reference counting later.

3.6 Port and Port Set Models

Ports are Fluke objects representing targets for IPC requests. Ports might represent CORBA or MOM objects, for example, or in Fluke implementations implementing full protection, they may be used as secure capabilities. A single server may maintain a number of ports, each distinguishing a particular object, client, or object/client pair.

Port sets are objects that act as a rendezvous point between client threads attempting to perform IPC to a port and server threads waiting for incoming IPC requests on one or more ports. Multiple ports can be attached to a single port set, and multiple server threads can wait for incoming requests on a single port set at once. This allows a single "pool" of service threads to service incoming IPC requests from many ports (i.e. requests associated with many "objects" the server exports).

Our model assumes that threads to connect to for IPC have been "found" (the purpose of ports, port sets, and references). Thus, models of ports or port sets were not implemented in detail.

1. `KR_USER_EXCEPTION` A processor exception occurred which should be blamed on the user (e.g. because the exception was generated while accessing user space). The current thread's exception_state contains the details.
2. `KR_PAGE_FAULT` Page fault occurred. This gets turned into a real KR_USER_EXCEPTION by the kentry layer if the page fault cannot be resolved in the kernel and no appropriate region keeper can be found to handle the fault
3. `KR_CANCEL` Another thread is trying to manipulate us and has asynchronously canceled us, e.g. due to thread_interrupt(), thread_get_state(), thread_set_state().
4. `KR_NO_MEMORY` Ran out of kernel memory.
5. `KR_RESTART` This return code indicates that we have context switched due to a wait, and we need to restart execution in user mode before doing anything else in case dependent things have changed.

Figure 2: The set of return codes used within the kernel

The only relevant interfaces used are `s_pset_wait_receive()` by servers and `s_port_capture()` by clients. The semantics of these operations have been distilled from the C code.

In our Promela implementation, we model these two functions to implement the interface and make use of as little state as possible. The rendezvous point is actually implemented as a bit in the server's state. If the rendezvous bit is set, it means that either the client or the server is waiting for a connection. So when a client wants to connect to a server, it checks the rendezvous bit to see if the server is waiting for a connection. If the server is waiting, the server is captured and the connection is made. If the server is not waiting for a connection, the client sets the rendezvous bit and waits for the server to notice it. When a server is waiting for an IPC connection, it checks the rendezvous bit and if a client is waiting, it makes the connection with the client. Otherwise the server sets the rendezvous bit and waits for the client. In either case the server blocks and is captured by the client which eventually establishes a connection with this server.

It is interesting to note that the majority of the bugs in our model came through this interface. Since we strayed from the actual implementation of the ports we missed subtle nuances like the server always being blocked, etc.

3.7 Kernel Entry Layer Model

A thread performing kernel operations is "controlled" by the kernel entry point and the return code. All kernel entry points are "clean." A thread that gets canceled will never reveal itself to be in the middle of a kernel operation; it will always appear to the canceling thread that it is about to begin a kernel operation, or that it is in user mode. To prevent long or complicated kernel functions from becoming a bottleneck, these operations are broken into sub-sequences.

For example the long `fluke_ipc_client_connect_send_over_receive()` operation that connects to a server, sends a request, waits for a reply and receives it into a buffer, can take quite a while. As portions of the operation are completed the kernel entrypoint is advanced. So, in this example, after the "connect" phase is completed, the kernel sets the thread's entrypoint to be `fluke_ipc_send_over_receive()`.

All of the major functions within the kernel return one of a small, well defined set of error codes. See Figure 2 for a complete list. The return code is used to signal special conditions. By convention, any function returning a non-zero return code signals that the kernel operation in progress has been canceled or must be restarted.

4 Results

In addition to the descriptive models, and a greater understanding of the Fluke IPC code, this project generated quite a bit of documentation of the kernel internals. We have static call traces of the IPC system from the user-level entry point down into the core kernel routines. We used fully cross-referenced html versions of the kernel code [6].

The most important results are those from the SPIN generated verifications of various scenarios. Because we modeled the kernel entry layer accurately we are able to generate simple test cases that are easy to code yet quite flexible.

A standard test case will enumerate the list of kernel entry-points it plans to use. The pre-processor will ensure that only the need parts of the Promela IPC implementation are included, dramatically reducing compile time by omitting all unused code. One proctype is created for each user-level thread. These proctypes can then "call" user-level fluke functions by macro expansion. Consider the basic send-and-receive test as an example:

```
do
:: TRUE ->
     sendData = 42;
     flukeClientConnectSend(fluke, server, sendData);
     assert(rc == 0);

     flukeClientDisconnect(fluke);
     assert(rc == 0);
od;
```

The `fluke` parameter is a handle to a proctype implementing the kernel. All of our functions implicitly set a global variable `rc`, Promela's `assert()` construct is then used to verify successful completion.

Certain simulation parameters are configurable through the use of preprocessor constants before including the Promela files that contain the definitions of the IPC model. At this point, there are three parameters which can be set. First, one can select whether or not a simulated transfer can generate a page fault. Every transfer can cause a page fault in either the sender, or the receive, or it cannot cause a page fault at all. This is controlled with `INCLUDE_IPC_TRANSFER_FAULTS`.

Secondly, `INCLUDE_IPC_PAYLOAD` determines whether IPC data payload should be included in the simulation. With this option, a single byte of data is transferred from the sender to the receiver. This lets us assert that data hasn't been altered during the IPC transport.

Lastly, it is possible to choose between two distinct models of mutexes in Promela (see section 3.2.) They differ in the number of checks and the amount of state associated with each. The "simple" mutexes are a minimal implementation without any consistency checks. The "safe" implementation includes safety checks (via `assert()`) and requires some extra state to be stored with each mutex. It is used if `INCLUDE_MUTEX_SAFE` is defined.

4.1 The Basic Send Test

This test, `src/test/Basic-send.pr`, is the first real test we consider here. It is is a very basic send-and-receive operation involving two threads, a client and a server. The client repeatedly connects, sends a byte of data, and disconnects. The server waits, receives the byte of data, disconnects and then starts over.

This tests the kernel entry layer, mutexes, the IPC connection logic; it also tests a bit of the rollback, as threads effectively rollback when moving from, for example, the "wait" to the "receive" in a "wait_receive" operation. If IPC page fault modeling is activated then arbitrary transfers will cause a page fault in the client or the receiver. Our verification runs show that the IPC system fully recovers from such faults.

A full verification run of this test takes about 41 MB of memory. Interestingly, it takes longer to compile the test than it does to run it. It generates 94,000 states, and only executes to a maximum depth of 1650. These are especially interesting results considering that over 60,000 lines of C code are generated by SPIN. Because the send-receive is a fairly lockstep operation–the server block until the client sends, the client blocks until the server acks, etc–it is easy to see why there is little interleaving among the states.

This test does not cover other "corner cases" than the interleaving of page faults in the code and the recovery steps involved, in particular it does not consider the case where either thread involved is canceled.

4.2 The Ack Test

This test, `src/test/Ack-test.pr`, exercises the acknowledgment portion of the protocol. Sending an ack tells the other half of your connection that you have finished receiving and are ready to send data back across the link.

This tests much of the code exercised by the basic send test, in addition the client send and ack-specific codes.

The scope of this test is similar to that of the basic send test, it requires approximately 20 MB of memory to run. Again, the compilation of the verifier takes longer than the verification itself.

This test has the same limitations as basic send, i.e., error conditions outside of a page fault are not considered. Still, all of the waiting, blocking and capturing occurs correctly in all instances of page-faults and all possible arrival ordering.

4.3 The Interposer Test

This is a more complicated test. It involves three threads. One is a client whose server thread is a client to a third thread. The thread in the middle is acting as an interposer–it receives data from the client and forwards it to the server.

This tests the middle thread's code path quite rigorously. It is acting as both a client and a server and can experience page faults on either side of its connection.

As a more complicated test, this would require about 400 MB of memory to hold all of its states. (Its has about 10 million states.) Supertrace lets us run it in 130 MB of memory.

4.4 The Cancel Test

This test, `src/test/Cancel-test.pr`, was our ultimate goal for the project. We hoped to show that the exportable kernel state (which is implemented via the thread cancel facility which allows a thread's state to be extracted at any time) was robust. It involves three threads. Two threads operate as client and server as in the basic send test, endlessly connecting and sending data. The third thread sits in an infinite loop and cancels either the client or the server.

While we haven't been able to run a complete successful verification of this test, we still believe our model to be sound. Currently the only known deadlock is encountered 900 steps into a verification. It is due entirely to a poor abstraction of ports and port sets. Specifically, we are not modeling the wait state, wait queue and cancelability of threads that are waiting to connect to their client or their server. The initial abstraction we implemented (a single bit for rendezvous) was too naieve. The IPC code expects the threads returned from a rendezvous to be in certain states. It also expects that the cancel pending flag will have been checked. This should be fixed as we continue to maintain and expand the system for the Flux project.

This test is quite large. A complete state space exploration would require on the order of 2 GB of memory. The supertrace option in SPIN lets it run in about 200 MB. The estimated coverage of the state space is 99.9%.

4.5 Other Tests

We developed our Promela in a bottom-up fashion, starting with the primitives used to implement the more complicated IPC operations. We have a full suite of tests that test our specific abstractions–wait queues, condition variables, mutexes, the kentry layer, etc.

4.6 Test Analysis

While debugging these tests, we have so far exclusively encountered bugs in our model of the system. It is interesting to note that none of these bugs have been in the parts of the model that were derived from the C code. They have all been in those areas of the model where the C code could not be literally translated into Promela, but where abstractions with equivalent semantics had to be reimplemented for efficiency and other reasons. For example, a bug in our model did not include checking for a pending cancel before blocking. This quickly broke the cancel test (i.e., within 100 steps.)

5 Conclusions

The results from this project were fairly typical of a small specification and verification effort: no major problems with the system being modeled were discovered (but we have more confidence that there are no problems), the specifiers gained insight into the workings of a complex part of the system, and we now have valuable documentation that should lower the learning curve for people trying to learn about Fluke internals.

In addition generating the documentation, and providing a foundation for further verification work, the project was successful in pointing out one latent bug. In implementing the Fluke security kernel, Steve Smalley added a new security-related wait state, without realizing that the wait states were organized in a hierarchy. The hierarchy in our documentation made him realize that the new wait state was wrong, and that he had made a lucky choice. So, the project has already had success in eliminating some ambiguity.

We plan to integrate much of the documentation developed under this project back into the Fluke kernel documentation and source code. In particular, Appendix A as well as the call traces contain information that will be valuable to anyone studying the kernel. We also have many useful comments that will be integrated back into the source code, such as the wait state hierarchy, etc. The Promela code itself may be valuable as encouragement for other OS teams to try out verification projects.

The Fluke IPC verification effort has seen a great deal of conceptual and organizational change in a very short amount of time. In particular, organizational techniques to promote Promela code understandability and to keep the state space size under control saw much evolution. Avenues which initially seemed viable, clean and attractive proved in the end to be difficult, unwieldy and self-defeating in often subtle and unexpected ways.

We've attempted to present in the preceding sections the organization that we currently find workable, and our optimism for its continued viability. We now seek to more fully justify this organization in comparison with other approaches. We first examine work by others, and discuss improvements we feel have been made over past approaches. We then move on to describe specific failed organizations that were attempted by this effort, and the problems we encountered.

5.1 Related Work

Two other interesting efforts have been made to verify the correctness of operating systems' IPC mechanisms. We examine these briefly here, paying particular attention to difficulties and inflexibilities in these other approaches which have been specifically addressed by the Fluke IPC verification effort.

5.1.1 Harmony

Cattel [1] performed a SPIN-based verification of the Harmony operating system, a "portable real-time multitasking multiprocess operating system" developed by the Canadian National Research Council in the 1980's and early 1990's. Cattel's effort sought to verify properties of Harmony as a whole, and attempted to model all kernel services. Like our effort, Cattel was concerned primarily with the interprocess synchronization mechanisms at work during IPC.

While Harmony IPC is less complex than Fluke IPC, there are structural similarities between the way Cattel structured his Promela and the way our Promela is structured. Cattel held IPC state in an array of "task descriptors" and pointed at them with their array index. We formalized this to the notion of the Reference, but the basic idea is the same. Also like our effort, Cattel generated his models by studying the implementation code of a pre-existing system. Thus, like our verification, Cattel had strong reason to believe that he was proving things about his particular implementation of the system, and not just its abstract specification.

However, Cattel seems to have been mainly concerned with showing freedom from deadlock, while we would like to consider stronger guarantees such as correct state progression in the presence of cancellation, as well as some sense of correct data delivery in this environment. Also, Cattel spends a great deal of effort analyzing and minimizing his scenario-space, with the implication that he would generate, by hand, a scenario for every possible interaction of sender and receiver. We seek to exploit SPIN's talents at exploring the scenario-space for us. We ultimately envision processes which nondeterministically try all allowed high-level operations, so that we are sure to explore all possible user errors, in addition to reasonable interleavings of operations.

5.1.2 RUBIS

Duval and Julliand utilitized SPIN to model the intertask communication facilities of the RUBIS microkernel [2]. Like our effort, theirs sought to build and exhaustively test usage scenarios. However, their effort placed emphasis on formally expressing mappings between implementation source code and Promela code via a collection of transformation rules. While these transformation rules do not appear to have been used to perform *automatic* transformation of source code to verifier code, the authors state their belief that such an automated transformation tool would be of great value.

We, on the other hand, question the success that such a tool might find. It is often the case that the verification of desired properties requires the addition of extra "hidden" state to a model, which represents some semantic state outside the implementation domain, but related to the property of interest. The ability to add this semantic state can make demands on the overall structure of the verification code. We have seen cases where semantic properties we wish to expose are distributed across much C code, for implementation reasons. In cases like these, it is unlikely that an automated tool would have the necessary intuition to extract these distributed semantics in a meaningful way.

We feel that a more plausible approach is the formulation of a meta-language, rich in explicit semantic content, from which both Promela code and implementation-language code could be automatically generated. We feel that such a meta-language could be made flexible enough to facilitate the evolution of software by allowing experimental manipulation of both semantic and implementation constructs, always in the presence of easy verification of desired properties.

The RUBIS modeling project also made a distinction between *abstract models* and *detailed models*. Abstract models are generated from high-level semantics defined in the system specification and are useful for formalizing the system functionality independently of implementation. Detailed models, on the other hand, are generated with the implementation of the system in mind, and can be used to verify that the system provides the behavior defined in the specification.

It is our belief that the rigidity of this distinction is unnecessary. The "sliding `d_step`" technique we employ for state-space management creates a spectrum of models from the abstract to the detailed. Placing `d_step`s around necessarily atomic segments in a minimal fashion creates a pure detailed model. Placing `d_step`s around high-level functions creates a pure abstract model. Since these models share the same code, we can be sure that they embody exactly the same semantics. The precise implementation of these semantics, once verified, can be abstracted away and the same Promela code can be used as the basis for constructing the next layer to be verified.

The RUBIS emphasis on abstract modeling at specification-time and detailed modeling at implementation-time ignores the complex and recursive interrelationship of these two processes. We believe the ability to slide `d_step`s lower and formally understand the semantic impact of late-cycle specification changes to be of great potential value to living software projects whose goals are in constant flux.

5.2 Lessons Learned

The progress of the Fluke IPC verification project thus far fills us with optimism. However, we've traveled down our share of blind alleys, and present brief summaries of these here:

- We were originally going to encapsulate the state associated with Thread objects as local state to a particular proctype, and access it by sending message to that proctype. The rationale was that the PO reductions allowed by encapsulating the state and declaring the communication channel "xs" to the "state-proc" would outweigh the added state-vector length that would result from the extra proctype instances. However, we found it impossible to get appropriate blocking semantics when locking Mutex'es held by this state-proc. Also, since local variables were required to hold values "gotten" from the state-proc (there is no stack), the state-vector costs were much greater than our original estimates. The approach was scrapped and thread state was made global, greatly simplifying all code that accessed it.

- **Fix:** *Any others?*

Also, we've noticed some potential improvements to the SPIN system itself:

- SPIN's `if` syntax is less than ideal for the task of translating nested if-then-else statements. Since "properly formatted" if :: :: fi statements take several lines, and the ":: else →" syntax requires deep indentation, it is tedious to translate C structures of the form `if ...else if ...else if ...`

- SPIN has internal hard-coded limits on how much code can be in d_step bodies. We hit some constant hidden in `d_step.c`, which we had to increase from 512 to 1024 to get our code to compile.

- Variable names (not necessarily variable slots) and labels should be scoped to their enclosing block, to allow goto's to be used inside macros that are instantiated multiple times in the same proctype.

- SPIN should understand hexadecimal integer constants, like `0xfff0`. Fluke "wait states" are even more cryptic in Promela than they were in the C code, since they must be expressed as decimal values. (Even the compilers generated by the 507 class do this!)

- Execution traces say "(1)" for macros whose bodies are skip, when the macro name is more meaningful (for example enabling and disabling interrupts). This is a manifestation of the larger problem that cpp'ed source is displayed in the traces, not "meaningful" source.

- A `printf()` is not available that can print during a verification. (Well, you can turn them *all* on, but if you only want one, its impossible.) This would be very helpful in generating coherent run-time error messages.

- The `assert()` should take a string argument explaining what went wrong. It would make assert failures far more informative.

- More flexible pre-processing support should be standard. In particular, SPIN should honor the CPP environment variable, or at least run the cpp on the user's PATH. (Currently the source hard codes `/usr/lib/cpp`.)

- Most SPIN error messages are completely inane. In particular, error messages that refer to nonexistent proctypes, error messages that report the ASCII values of trouble characters, and messages that say "*thingX* seen near *thingX*" are particularly frustrating.

- Accurate reporting of error line number and source file would be useful.

Several of the above items result from a fundamental tension between the simplicity that Promela's designers want you to focus on, and the complexity of Fluke IPC. Promela does not have functions, partly in order to make designers think hard about making the specification as simple as possible. In order to adequately capture Fluke functionality and keep the Promela code readable, we rely heavily on a preprocessor to provide the illusion of function calls. That made debugging difficult, but the only other choice would be to duplicate code in many different places.

Finally, a note on the basic approach we took. Although the use of Promela/SPIN was effective in analyzing a reverse engineered specification of the Fluke IPC path, we do not believe that SPIN would have been as effective in developing a from-scratch specification. It is certainly possible to develop a specification (in a natural style for Promela), but it is very unlikely that the corresponding implementation could be so easily related back to the specification. Also, the Promela specification is of limited value in developing alternate (e.g., highly optimized) versions of Fluke IPC.

5.3 Prospects for the Future

In conclusion, we feel that the work completed thus far serves a dual purpose. First, it represents a successful proof-of-concept that SPIN- based model-checking is a viable approach for verifying certain properties about the Fluke operating system. Second, it lays the groundwork for the verification of many properties of the Fluke IPC system. In addition, we are optimistic that the techniques learned thus far could be applied toward other facets of the Fluke operating system.

A Appendix

This section contains details about the Fluke C implementation of the IPC mechanism.

A.1 Functions implementing components of reliable IPC paths

The implementation makes use of a set of core functions listed below. Most paths are built from two phases: 1(send) and 2(receive). All phase 1 routines are called with the current thread in an unknown state, and on successful return, the other thread is captured as the receiver. All phase 2 routines are called with the other thread captured as receiver, and return with everything all done, ready to return from the syscall. Routines that implement both phases 1 and 2 are marked _12_.

A.2 Client-side reliable IPC path components

- ipc_client_1_connect_send(client_thread, connected_ip)
  First disconnect any connection client_thread is involved in. Establish a connection to a new server thread using s_port_capture(). Set the server field of the client and the client field of the server appropriately and change the flags of the client and the server to show that the client is now a sender and the server is no longer a sender. Transfer the minimum message from the client to the server. Set the IP field of the client to the connected_ip parameter and the IP field of the server to fluke_ipc_server_receive__entry. Then perform a reliable transfer operation from the client to the server.

- ipc_client_1_ack_send(client_thread, acked_ip)
  First make sure the connection is active (by calling ipc_client_find_server). If the server is not waiting as a sender (i.e. it's wait state is not at least WAIT_IPC_SRV_SENDER) put the client thread to sleep in the WAIT_IPC_CLI_ASEND state and return. If the server is waiting as a sender, capture the server. If the server wasn't waiting in WAIT_IPC_SRV_ORECV (i.e. it is not waiting for a over and receive) put the client thread to sleep in the wait state WAIT_IPC_CLI_ASEND, put the server on the ready queue (by calling thread_handoff) and return. If the above condition doesn't hold this means both the client and the server are ready to reverse. Change the flags of the server and the client to indicate that the client will now

be a sender and that the server is no longer a sender. Transfer minimum message from the client to the server. Set the IP of the client to the acked_ip parameter and that of the server to fluke_ipc_server_receive__entry. Now perform a reliable transfer from the client to the server.

- ipc_client_1_send(client_thread, out_wval)
  First make sure the connection is active (by calling ipc_client_find_server). If the server is not waiting as a receiver (i.e. it's wait state is not at least WAIT_IPC_SRV_RECEIVER) put the client_thread to sleep in the wait state WAIT_IPC_CLI_SEND and return. Otherwise capture the server as a receiver. If the server was waiting to receive (i.e. in the wait state WAIT_IPC_SRV_RECV) do a reliable transfer from the client to the server. If the server was waiting in the wait state WAIT_IPC_SRV_ASEND, this means that the server is acking the client before it is done and so just throw away the send data.

- ipc_client_2_over_receive(client_thread, wval)
  If the wval parameter is WAIT_IPC_SRV_RECV, this means that the server still thinks it is receiving data and so notify it that the message has ended using a IPC_FINISH_RECEIVE. Put the client to sleep in the wait state WAIT_IPC_CLI_ORECV and put the server on the ready queue.

- ipc_client_12_receive(client_thread)
  First make sure the connection is active (by calling ipc_client_find_server). If the server is not waiting as a sender (i.e. it's wait state is not WAIT_IPC_SRV_SENDER), put the client thread to sleep in the wait state WAIT_IPC_CLI_RECV (by calling thread_wait) and return. Otherwise capture the server thread as the sender. If the the server is not waiting to send (i.e. it's wait state is not WAIT_IPC_SRV_SEND, it means that the server is done sending; so release the server, notify the client that the message has ended using a IPC_FINISH_RECEIVE and return KR_RESTART. Otherwise put the client to sleep in the wait state WAIT_IPC_CLI_RECV and release the server.

A.3 Server-side reliable IPC path components

- ipc_server_1_ack_send(server_thread, acked_ip)
  Similar to ipc_client_1_ack_end() described above.

- ipc_server_1_send(server_thread, out_wval)
  Similar to ipc_client_1_send() described above.

- ipc_server_2_wait_receive(server_thread, wval)
  Break the connection with the client thread and release it. Nuke the client field of the server and the server field of the client. Do a FINISH_RECEIVE, set the IP to fluke_ipc_wait_receive__entry put the client on the ready queue and then wait on a port set for an incoming IPC request.

- ipc_server_2_over_receive(server_thread, wval)
  Similar to ipc_client_2_over_receive() described above.

- ipc_server_3_wait_receive(server_thread)
  Set the IP of the server to fluke_ipc_wait_receive__entry and wait on a port set for an incoming IPC request.

- ipc_server_12_receive(server_thread)
  Similar to ipc_client_12_receive() described above.

A.4 IPC machine dependent macros

`EXC_SET_IP(ip)` and `EXC_GET_IP()` are wrappers that set the machine depenedent instruction pointer. They are used for setting a new kernel entry point. For example, during a `fluke_client_connect_send_over_receive()`, after the "connect" phase is complete, the kernel will set the IP to be the `fluke_send_over_receive()` entry point. To exit the kernel, the IP to `fluke_nop__entry`.

`IPC_FINISH_RECEIVE(thread, status)`. This macro sets the user-return code for a kernel syscall, and sets the thread's IP to `fluke_nop__entry`. It also does some silly stuff with the "min_msg" registers–but only because they aren't real registers.

`IPC_STATUS(thread)` sets the status code for the current ipc syscall. The details are quite grungy: On the x86 the eax register is overloaded. The bottom 16 bits are the number of buffers the invoker is using (an in/out parameter), the top 16 bits are the return code of the IPC call.

`EXC_RETURN_INSANITY(cond)` set the `exc_state.code` to the given insanity condintion, and returns a KR_USER_EXCEPTION.

A.5 IPC Pickle Operations

- `void ipc_pickle(s_thread_t *client, s_thread_t *server)`

  Takes client and server threads. They must be pointing to each other respectively. Moves the active pointers into indirect links. Zero's out the direct pointers.

- `void ipc_unpickle(s_thread_t *client, s_thread_t *server)`

  Destroy the indirect pointers. (We don't have to "follow" them, because they were passed in as parameters.)

  Check that the two threads agree on the "direction" of the connection–

  The server's direct client pointer is set to the client and vice-versa-vice.

- `int ipc_client_unpickle(s_thread_t *client, int no_wait)`

  Client must be the current thread. It must have no direct server pointer. (Its trying to restore the direct pointer.)

  First, the indirect server link is turned into a direct pointer. If the link is bogus, 0 is returned. (The object is locked (the `sob` lock) if the link was successfully followed.) If the link pointed to a non-thread object, then unlock the object and return.

  If the server is in WAIT_IPC_UNPICKLE, meaning that it is waiting for its "other half" to pickle a connection, then we unlock the server. The thread_wakeup() call put the thread in the WAIT_NONE state, and its not on any queues, so we have it "captured".

  If the captured server's *passive client link* isn't pointing at us, then the client blocks in WAIT_IPC_PICKLE and hands off to the server. (An exception being, that if the no_wait flag is passed in non-zero, then the server is readied and we return immediately without blocking.) Otherwise, call `ipc_unpickle(client, server)` which will restore the active links. Then ready the server, as we had it captured.

  But, if the server wasn't in WAIT_IPC_UNPICKLE, then if we're the no_wait flag was passed in non-zero,

- `int ipc_server_unpickle(s_thread_t *server, int no_wait)`

A.6 Thread Wakeup and Ready Functions

These are the functions used to put threads to sleep, wake them up and manipulate their wait state.

- `thread_wakeup(target_thread, required_state)`

Disables interrupts, spin locks 'target_thread's wait_state lock and then calls `id_thread_wakeup_locked` which wakes up and captures 'target_thread' iff its current wait value is *at least* 'required_state'. (See the Wait State section for details on the wait states.)

If 'target_thread' is waiting in `WAIT_ON_COND` state, then it is removed from the queue and returned to the WAIT_NONE state. The old wait state of the target thread is returned. (Note that there is a potential race condition handled in here, too. If a thread is woken from a condition variable wait, the kernel can't change its wait state until after its been removed from the wait queue. Thus in this code, if the thread is waiting on a condition variable, but is not actually on the specified list, then it is in the process of being removed by `id_cond_wakeup()` which has removed it from the list, but has not updated its state.)

If 'target_thread' is not in `WAIT_COND` then 'target_thread's old wait state is returned and it is put into the WAIT_NONE state.

If 'target_thread' is not in at least 'required_state' then 0 is returned.

- `thread_handoff(current_thread, state, target_thread)`

  Give control to 'target_thread' atomically, put ourselves in waiting 'state', and handle the possibility of us getting canceled.

  Disable interrupts, acquire spinlocks on both wait states in defined order to avoid deadlock, make sure that our own wait state is `WAIT_NONE`. Then, put 'target_thread' on the ready queue using `id_thread_ready` (because it's locked), release the lock and let the other thread go. That's all that's done to the other thread.

  At this point, a cancel may be already pending (indicated by `wait_state.cancel_pending`). If one is pending, then there are two cases:

  – The state we want to wait in is *not cancelable* (i.e., it doesn't have `WAIT_CANCELABLE` in its pattern - unlike most of the intermediate IPC states) - then set our resume_rc to `KR_CANCEL`. Note that we don't return here. This is a somewhat special case.

  – If, however, the state we should wait in is *cancelable*, then unlock the current thread object, enable interrupts again and return immediately with `KR_CANCEL`. Don't context switch. Consider yourself canceled.

  At this point, resume_rc is either set to `KR_RESTART` or to `KR_CANCEL`, depending on whether we have a cancelation pending or not. We now set the current thread's wait state to 'state' and go to the dispatcher.

  Cryptic comment #21a in the code says:

```
/*
 * We don't need to add the current thread to any wait queue,
 * because the wait state itself defines the "ownership" of the thread.
 */
```

  This simply means that another thread will explicitly point at us and make us ready - rather than we being one thread among dozens waiting for an event to happen.

  Eventually, the dispatcher will reschedule us (because someone else put us on the ready queue), and then we reenable interrupts and return with either `KR_RESTART` or `KR_CANCEL`, depending on whether a cancelation was pending or not when we invoked the dispatcher.

  Note that the "unusual" case, where the dispatcher is invoked even though we already knew a cancelation was pending is only used when actually handling a cancelation.

- `thread_wait(current_thread, state, other_thread, other_state)`

Put current_thread to sleep in the given state, (like thread_handoff) unless the other_thread is already in other_state. Perhaps perform an atomic action on other_thread before we go to sleep.

This function is similar to thread_handoff() above (80% of the code is identical), but instead of simply handing off to another thread, it checks to see if other_thread has reached other_state. If so, thread_wait returns immediately with `KR_RESTART`. Otherwise, the current_thread is put to sleep in wait_state state.

The function first disables interrrupts, and grabs spin locks for both of the threads' wait_states.

If other_state is equal to 0, the other_thread is canceled before current_thread is put to sleep. Also, if other_state is equal to -1, it is a magic flag to unlock the other_thread->sob.lock. Both of these actions must be performed atomically due to potential race conditions with the other_thread waking up before current_thread is really asleep.

At this point, a cancel may be already pending (indicated by `wait_state.cancel_pending`). If one is pending, then there are two cases:

- The state we want to wait in is *not cancelable* (i.e., it doesn't have `WAIT_CANCELABLE` in its pattern - unlike most of the intermediate IPC states) - then set our resume_rc to `KR_CANCEL`. Note that we don't return here. This is a somewhat special case.
- If, however, the state we should wait in is *cancelable*, then unlock the current thread object, enable interrupts again and return immediately with `KR_CANCEL`. Don't context switch. Consider yourself canceled.

At this point, resume_rc is either set to `KR_RESTART` or to `KR_CANCEL`, depending on whether we have a cancelation pending or not. We now set the current thread's wait state to 'state' and go to the dispatcher.

Cryptic comment #21a in the code says:

```
/*
 * We don't need to add the current thread to any wait queue,
 * because the wait state itself defines the "ownership" of the thread.
 */
```

This simply means that another thread will explicitly point at us and make us ready - rather than we being one thread among dozens waiting for an event to happen.

Eventually, the dispatcher will reschedule us (because someone else put us on the ready queue), and then we reenable interrupts and return with either `KR_RESTART` or `KR_CANCEL`, depending on whether a cancelation was pending or not when we invoked the dispatcher.

Note that the "unusual" case, where the dispatcher is invoked even though we already knew a cancelation was pending is only used when actually handling a cancelation.

- `thread_cancel(thread)`

  Set the cancel pending bit in thread. Disables interrupts, locks the wait_state.lock, and sets thread.cancel_pending. If it can wakeup the thread, it does so, sets the thread's resume_rc to KR_CANCEL and readies it.

References

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

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

[3] 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.

[4] 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*, Seattle, WA, Oct. 1996. USENIX Assoc.

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

[6] `makehtml`. The source code to the fluke operating system written in C (HTML format). file://home/css/grad/gback/CVS/flukehtml/kernel/main.c.html, 1996.

[7] 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*, Seattle, WA, Oct. 1996. IEEE.